

# Effectful composition in natural language semantics

Effect-driven interpretation

Dylan Bumford (UCLA)   Simon Charlow (Yale)

NASSLLI 2025 @ UW

June 25, 2025

# Functors

## Some natural choices for effect types

Expression	Type	Denotation
it	$i \rightarrow e$	$\lambda i. \pi i$ (e.g., $\lambda i. i$ )
the planet	$e + \perp$	$x$ if <b>planet</b> = $\{x\}$ else #
Jupiter, a planet	$e \times t$	$\langle j, \mathbf{planet}j \rangle$
which planet	$\{e\}$	$\{x \mid \mathbf{planet}x\}$
no planet	$(e \rightarrow t) \rightarrow t$	$\lambda Q. \neg \exists x. \mathbf{planet}x \wedge Qx$
JUPITER	$e \times \{e\}$	$\langle j, \{x \mid x \sim j\} \rangle$
as for Jupiter	$s \rightarrow (e \times s)$	$\lambda s. \langle j, j \# s \rangle$
a planet	$s \rightarrow \{e \times s\}$	$\lambda s. \{ \langle x, x \# s \rangle \mid \mathbf{planet}x \}$

It's worth considering what it would take to develop a standard grammar by hand to deal with all these effects. It would be complex, inflexible, brittle, and arbitrary.

# Functors

In each case, we have an **e** situated in some kind of **structural context**

---

it	$i \rightarrow e$
the planet	$e + \perp$
Jupiter, a planet	$e \times t$
which planet	$\{e\}$
no planet	$(e \rightarrow t) \rightarrow t$
JUPITER	$e \times \{e\}$
as for Jupiter	$s \rightarrow e \times s$
a planet	$s \rightarrow \{e \times s\}$

---

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

# Functors

In each case, we have an **e** situated in some kind of **structural context**

---

it	$R\mathbf{e} ::= i \rightarrow \mathbf{e}$
the planet	$M\mathbf{e} ::= \mathbf{e} + \perp$
Jupiter, a planet	$W\mathbf{e} ::= \mathbf{e} \times t$
which planet	$S\mathbf{e} ::= \{\mathbf{e}\}$
no planet	$C\mathbf{e} ::= (\mathbf{e} \rightarrow t) \rightarrow t$
JUPITER	$F\mathbf{e} ::= \mathbf{e} \times \{\mathbf{e}\}$
as for Jupiter	$T\mathbf{e} ::= s \rightarrow \mathbf{e} \times s$
a planet	$D\mathbf{e} ::= s \rightarrow \{\mathbf{e} \times s\}$

---

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

# Functors

In each case, we have an **e** situated in some kind of **structural context**

---

it	$R \blacksquare ::= i \rightarrow \blacksquare$
the planet	$M \blacksquare ::= \blacksquare + \perp$
Jupiter, a planet	$W \blacksquare ::= \blacksquare \times t$
which planet	$S \blacksquare ::= \{\blacksquare\}$
no planet	$C \blacksquare ::= (\blacksquare \rightarrow t) \rightarrow t$
JUPITER	$F \blacksquare ::= \blacksquare \times \{\blacksquare\}$
as for Jupiter	$T \blacksquare ::= s \rightarrow \blacksquare \times s$
a planet	$D \blacksquare ::= s \rightarrow \{\blacksquare \times s\}$

---

These contexts embody different sorts of **enrichments** to meaning. They are known in Category- and Programming- Theory as **Functors**

## Functors as **boxes**

Functors are boxes. We can open the box and mess with the contents. Given some  $f : e \rightarrow \tau$ , we can readily derive apply  $f$  “inside” our enriched e’s:

$$\blacktriangleright \{x \mid \mathbf{planet} x\} \xrightarrow{\cdot f} \{f x \mid \mathbf{planet} x\}$$

S  $\tau$

## Functors as boxes

Functors are boxes. We can open the box and mess with the contents. Given some  $f : e \rightarrow \tau$ , we can readily derive apply  $f$  “inside” our enriched e’s:

- ▶  $\{x \mid \mathbf{planet}x\} \xrightarrow{\bullet f} \{f x \mid \mathbf{planet}x\}$  S $\tau$
- ▶  $\langle j, \mathbf{planet}j \rangle \xrightarrow{\bullet f} \langle f j, \mathbf{planet}j \rangle$  W $\tau$

## Functors as boxes

Functors are boxes. We can open the box and mess with the contents. Given some  $f : e \rightarrow \tau$ , we can readily derive apply  $f$  “inside” our enriched e’s:

- ▶  $\{x \mid \mathbf{planet}x\} \xrightarrow{\bullet f} \{f x \mid \mathbf{planet}x\}$  S $\tau$
- ▶  $\langle j, \mathbf{planet}j \rangle \xrightarrow{\bullet f} \langle f j, \mathbf{planet}j \rangle$  W $\tau$
- ▶  $x \text{ if } \mathbf{planet} = \{x\} \text{ else } \# \xrightarrow{\bullet f} f x \text{ if } \mathbf{planet} = \{x\} \text{ else } \#$  M $\tau$

## Functors as boxes

Functors are boxes. We can open the box and mess with the contents. Given some  $f : e \rightarrow \tau$ , we can readily derive apply  $f$  “inside” our enriched e’s:

- ▶  $\{x \mid \mathbf{planet} x\} \xrightarrow{\cdot f} \{f x \mid \mathbf{planet} x\}$  S  $\tau$
- ▶  $\langle j, \mathbf{planet} j \rangle \xrightarrow{\cdot f} \langle f j, \mathbf{planet} j \rangle$  W  $\tau$
- ▶  $x \text{ if } \mathbf{planet} = \{x\} \text{ else } \# \xrightarrow{\cdot f} f x \text{ if } \mathbf{planet} = \{x\} \text{ else } \#$  M  $\tau$
- ▶  $\lambda x. x \xrightarrow{\cdot f} \lambda x. f x$  R  $\tau$

## Functors as boxes

Functors are boxes. We can open the box and mess with the contents. Given some  $f : e \rightarrow \tau$ , we can readily derive apply  $f$  “inside” our enriched e’s:

- ▶  $\{x \mid \mathbf{planet}x\} \xrightarrow{\bullet f} \{f x \mid \mathbf{planet}x\}$  S $\tau$
- ▶  $\langle j, \mathbf{planet}j \rangle \xrightarrow{\bullet f} \langle f j, \mathbf{planet}j \rangle$  W $\tau$
- ▶  $x \text{ if } \mathbf{planet} = \{x\} \text{ else } \# \xrightarrow{\bullet f} f x \text{ if } \mathbf{planet} = \{x\} \text{ else } \#$  M $\tau$
- ▶  $\lambda x. x \xrightarrow{\bullet f} \lambda x. f x$  R $\tau$
- ▶  $\langle j, \{x \mid x \sim j\} \rangle \xrightarrow{\bullet f} \langle f j, \{f x \mid x \sim j\} \rangle$  F $\tau$
- ▶  $\lambda s. \{ \langle x, s ++ x \rangle \mid \mathbf{planet}x \} \xrightarrow{\bullet f} \lambda s. \{ \langle f x, s ++ x \rangle \mid \mathbf{planet}x \}$  D $\tau$
- ▶  $\lambda c. \neg \exists x. \mathbf{planet}x \wedge c x \xrightarrow{\bullet f} \lambda c. \neg \exists x. \mathbf{planet}x \wedge c(f x)$  C $\tau$
- ▶ ...

# Functors formally and in Haskell

For any functor  $F$ , we have  $\bullet : (a \rightarrow b) \rightarrow (F a \rightarrow F b)$ , such that:

- ▶  $\text{id} \bullet M = M$
- ▶  $(f \circ g) \bullet M = f \bullet (g \bullet M)$

**Identity**  
**Composition**

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

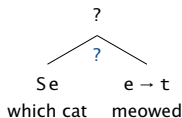
```
data Maybe a = Just a | Nothing deriving Functor
-- fmap f Nothing = Nothing
-- fmap f (Just x) = Just (f x)
```

```
data Reader i a = Reader (i -> a) deriving Functor
-- fmap f (Reader r) = Reader (\i -> f (r i))
```

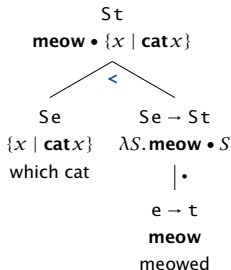
```
data Writer w a = Writer w a deriving Functor
-- fmap f (Writer w x) = Writer w (f x)
```

## Composition again

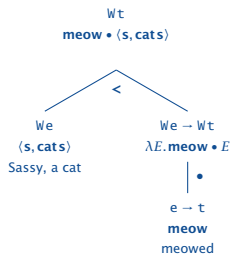
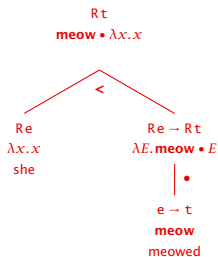
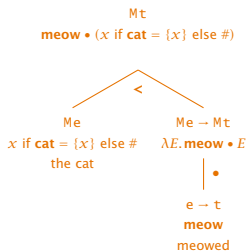
These effectful bits of language need to be able to slot in where no effect is expected



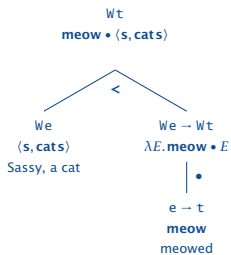
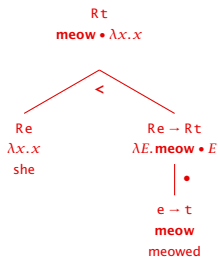
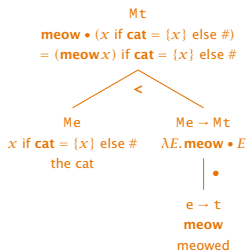
How does knowing that  $S$  is a Functor help? Well, we can now apply `fmap` to turn the VP into a function expecting an  $Se$  instead of an ordinary  $e$



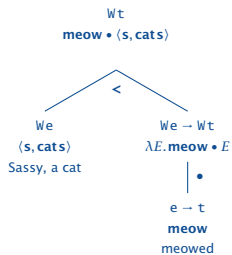
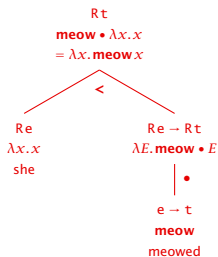
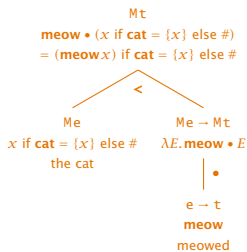
# Pleasingly uniform



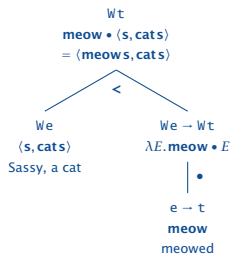
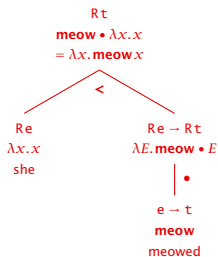
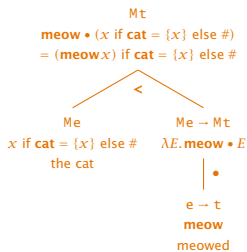
# Pleasingly uniform



# Pleasingly uniform

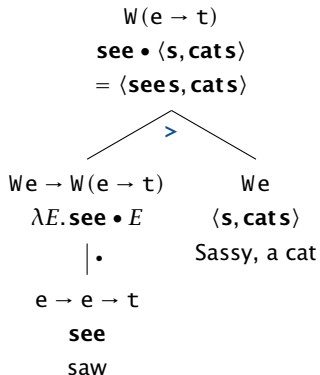


# Pleasingly uniform



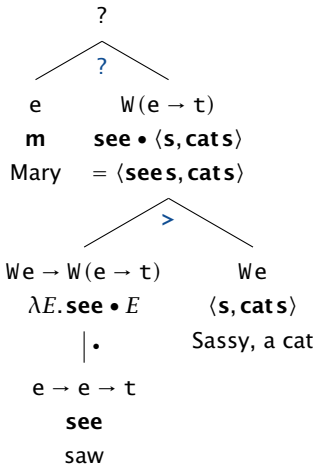
## Composition in more positions

Composing a verb with an effectful object is as easy as with an effectful subject



## Composition in more problematic positions

However, there *is* a problem combining an effectful VP with an ordinary subject:



(•) combines an *ordinary function* with an *effectful argument*; this is the opposite

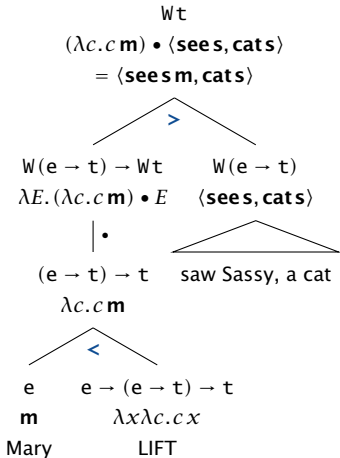
## Lifting

To proceed, we might simply lean on the oldest trick in the semanticist's book: invert the function-argument relationship

Expression	Type	Denotation
LIFT	$a \rightarrow (a \rightarrow b) \rightarrow b$	$\lambda x \lambda c. c x$

With this, the ordinary argument becomes the ordinary function, and the effectful function becomes the effectful argument

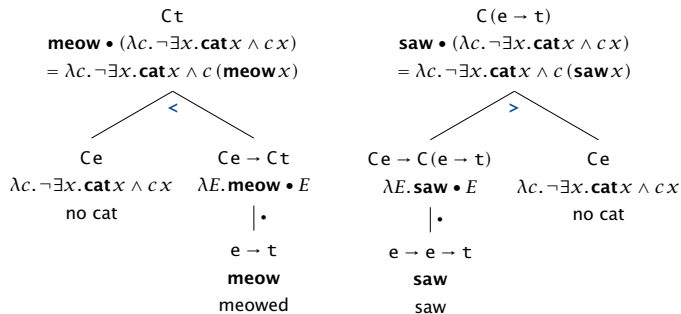
So we **fmap** once more



# Percolation

With what we have so far, it's easy to see that an effectful type anywhere in a derivation taints everything above it (the effect **percolates upward**)

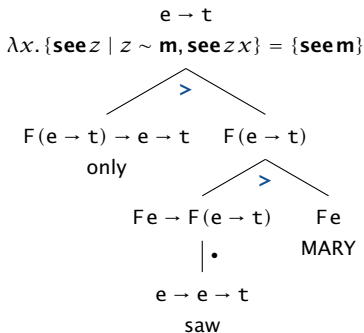
Particularly eyebrow-raising perhaps is the case of quantificational expressions



## Association with effects

In some cases, there are expressions that **associate with effects**, taking an effectful meaning as argument and returning something pure

Expression	Type	Denotation
only	$F(e \rightarrow t) \rightarrow e \rightarrow t$	$\lambda\langle P, C \rangle \lambda x. \{Q \in C \mid Qx\} = \{P\}$



## Types ending in $\mathfrak{t}$

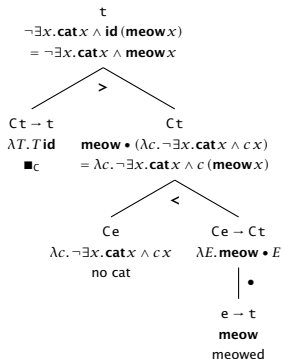
In other cases, a truth value may be extracted from an effectful meaning in virtue of some broader **linking hypothesis** about how the data structure relates to truth.

These extraction procedures are sometimes called **closure**, or **lowering**, operators, which we might write  $\blacksquare_H : H \mathfrak{t} \rightarrow \mathfrak{t}$ .

- ▶ A sentence with a supplement is true only if both of its dimensions are true (cf. Boër & Lycan 1976)  
 $\blacksquare_W = \lambda \langle p, q \rangle. p \wedge q$
- ▶ A sentence with a presupposition is true only if it is defined and not false (cf. the *A*-sassertion operator of trivalent logics like Beaver & Krahmer 2001)  
 $\blacksquare_M = \lambda m. \text{false if } m = \# \text{ else } m$
- ▶ A sentence that evokes many alternatives is true only if one of them is true (cf. Existential Closure, as in Kratzer & Shimoyama 2002)  
 $\blacksquare_S = \lambda S. \bigvee S$

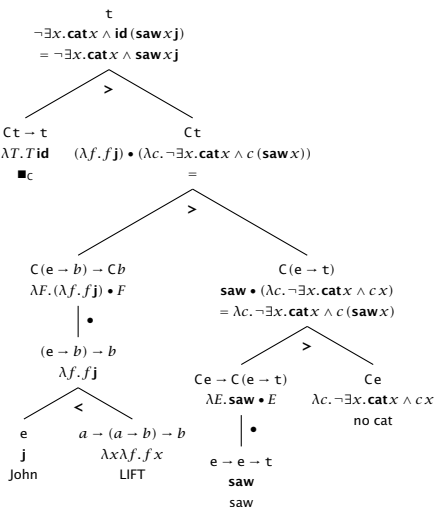
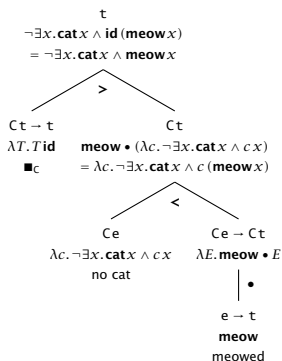
## Closing over continuations

For our scope-taking effect  $C$ , the standard closure operator is to run the denotation with a trivial identity continuation (Barker 2002):  $\blacksquare_C = \lambda T. T \mathbf{id}$



## Closing over continuations

For our scope-taking effect  $C$ , the standard closure operator is to run the denotation with a trivial identity continuation (Barker 2002):  $\blacksquare_C = \lambda T. T \text{id}$



Effect-driven interpretation

## Multiple pronouns

Let's consider some slightly more complex cases.

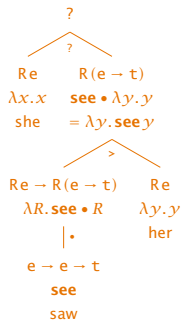
- ▶ John said she saw her.

What should the meaning of this expression be? Two pronouns, two requests!

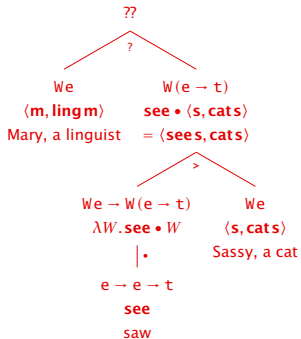
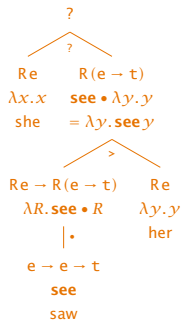
- ▶ she saw her :  $R(Rt)$   
[[she saw her]] =  $\lambda x.\lambda y.$  **saw**  $y x$

Can we derive something of this type, using our existing apparatus?

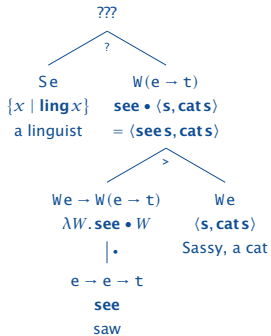
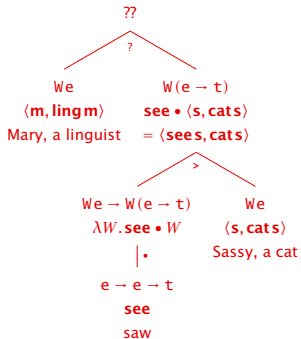
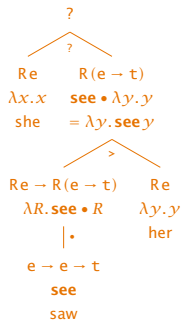
## Some puzzling configurations



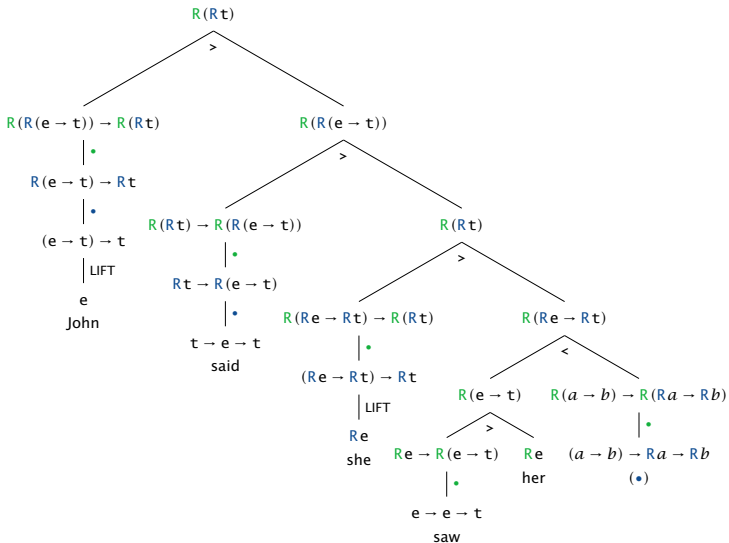
## Some puzzling configurations



# Some puzzling configurations



# Yes, but (Jacobson 1999, Charlow 2014)





## Comments, questions, concerns

Are silent occurrences of • actually instantiated in the syntax?

How is this *practical*? LIFT and • can apply iteratively. How would we ever know when to stop, or that a new (or first) analysis was not around the next corner?

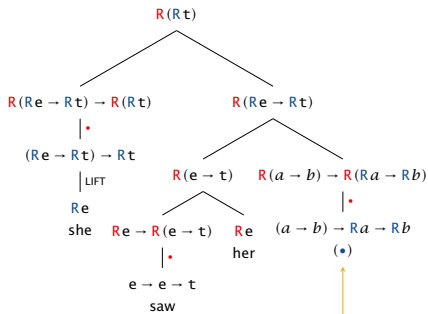
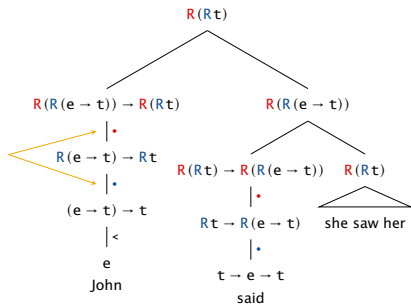
The system is unwieldy. Derivations are difficult to construct and significantly more complex than the readily justifiable syntax.

We would prefer a solution much more in the vein of **direct-style** treatments of effects in programming languages, but with referential transparency

What about PM, etc? How does • help us with, e.g., *dog near her*?

# Recursion

But much of the expressive power comes from allowing these combinators to be **partially applied** and/or **applied to themselves**



# Type-driven interpretation

$(>) \quad : (\sigma \rightarrow \tau) \rightarrow \sigma \rightarrow \tau$

$f > x := f x$

**Forward Application**

$(<) \quad : \sigma \rightarrow (\sigma \rightarrow \tau) \rightarrow \tau$

$x < f := f x$

**Backward Application**

$(\circ) \quad : (\beta \rightarrow \zeta) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \zeta$

$f \circ g := \lambda x. f (g x)$

**Function Composition**

$(\sqcap) \quad : (\sigma \rightarrow \mathbf{t}) \rightarrow (\sigma \rightarrow \mathbf{t}) \rightarrow \sigma \rightarrow \mathbf{t}$

$f \sqcap g := \lambda x. f x \wedge g x$

**Predicate Modification**

$(\upharpoonright) \quad : (\sigma \rightarrow \tau \rightarrow \mathbf{t}) \rightarrow (\sigma \rightarrow \mathbf{t}) \rightarrow \sigma \rightarrow \tau \rightarrow \mathbf{t}$

$r \upharpoonright p := \lambda x. \lambda y. p x \wedge r x y$

**Relation Restriction**

## Extending type-driven interpretation (Bumford & Charlow 2025)

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{\text{[redacted]} : Fc}{l : a \quad r : Fb} \bar{F}*$$

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{\text{[redacted]} : Fc}{l : Fa \quad r : b} \bar{F}*$$

Effect-driven interpretation (informally)

If  $a * b \vdash c$ , then  $a * Fb \vdash Fc$ , and  $Fa * b \vdash Fc$ !

## Extending type-driven interpretation (Bumford & Charlow 2025)

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{(\lambda r'. l * r') \bullet r : Fc}{l : a \quad r : Fb} \bar{F}*$$

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{(\lambda l'. l' * r) \bullet l : Fc}{l : Fa \quad r : b} \bar{F}*$$

Effect-driven interpretation (informally)

If  $a * b \vdash c$ , then  $a * Fb \vdash Fc$ , and  $Fa * b \vdash Fc$ !

## Some combinations

For any applicatives  $F$ ,  $G$ , if  $a * b \vdash c$ , then:

- ▶  $a * Gb \vdash Gc$   $\vec{F}*$
- ▶  $Fa * Gb \vdash F(Gc)$   $\vec{F}(\vec{F}*)$

The reverse direction works as well:

- ▶  $Fa * b \vdash Fc$   $\vec{F}*$
- ▶  $Fa * Gb \vdash G(Fc)$   $\vec{F}(\vec{F}*)$

If  $F$  and  $G$  are the same, we may derive **higher-order** meanings, corresponding (e.g.) to both layerings of effects in *a doctor examined every patient*.

## Recurring on the right

$$\frac{fx : t}{f : e \rightarrow t \quad x : e} >$$

[1st-order MoC]

$$\frac{f \bullet m : Rt}{(\lambda x. fx) \bullet m : Rt} =$$
$$\frac{f : e \rightarrow t \quad m : Re}{\bar{F} >}$$

[2nd-order MoC]

$$\frac{(\lambda m. f \bullet m) \bullet M : R(Rt)}{f : e \rightarrow t \quad M : R(Re)} \bar{F}(\bar{F} >)$$

[3rd-order MoC]

## Recurring on the right

$$\frac{fx : t}{f : e \rightarrow t \quad x : e} >$$

[1st-order MoC]

$$\frac{f \bullet m : Rt}{(\lambda x. fx) \bullet m : Rt} =$$

$$\frac{f : e \rightarrow t \quad m : Re}{\bar{F} >}$$

[2nd-order MoC]

$\lambda u. \lambda v. f(Muv)$



$$\frac{(\lambda m. f \bullet m) \bullet M : R(Rt)}{f : e \rightarrow t \quad M : R(Re)} \bar{F}(\bar{F} >)$$

[3rd-order MoC]

## Recurring on the left

$$\frac{f x : t}{f : e \rightarrow t \quad x : e} >$$

[1st-order MoC]

$$\frac{\text{LIFT } x \bullet m : R t}{(\lambda f. f x) \bullet m : R t} =$$
$$\frac{m : R(e \rightarrow t) \quad x : e}{\text{F}} >$$

[2nd-order MoC]

$$\frac{(\lambda m. \text{LIFT } x \bullet m) \bullet M : R(R t)}{M : R(R(e \rightarrow t)) \quad x : e} \text{F}(\text{F} >)$$

[3rd-order MoC]

## Recurring on the left

$$\frac{f x : t}{f : e \rightarrow t \quad x : e} >$$

[1st-order MoC]

$$\frac{\text{LIFT } x \bullet m : R t}{(\lambda f. f x) \bullet m : R t} =$$

$$\frac{m : R(e \rightarrow t) \quad x : e}{\bar{F} >}$$

[2nd-order MoC]

$\lambda u. \lambda v. (M u v) x$

$$\frac{(\lambda m. \text{LIFT } x \bullet m) \bullet M : R(R t)}{M : R(R(e \rightarrow t)) \quad x : e} \bar{F}(\bar{F} >)$$

[3rd-order MoC]

## Recurring on both sides

When **both** daughters are functorial, **both** rules apply, so we have an ambiguity

$$\frac{(\lambda r. (\lambda l. l * r) \bullet L) \bullet R : GFv}{L : Fs \quad R : Gu} \vec{F}(\vec{F}*)$$

$$\frac{(\lambda l. (\lambda r. l * r) \bullet R) \bullet L : FGv}{L : Fs \quad R : Gu} \vec{F}(\vec{F}*)$$

## Functors compose

From the functoriality of  $F$  and  $G$  alone, we may deduce the functoriality of  $F \circ G$ :

$$(\bullet)((\bullet)f) :: F(Ga) \rightarrow F(Gb)$$
$$| \bullet$$
$$(\bullet)f :: Ga \rightarrow Gb$$
$$| \bullet$$
$$f :: a \rightarrow b$$

```
ghci> :t fmap . fmap
fmap . fmap
  :: (Functor f1, Functor f2) =>
     (a -> b) ->
     f1 (f2 a) ->
     f1 (f2 b)
```

In other words, the composite fmap  $(\bullet)$  is given by  $(\bullet) \circ (\bullet)$ !

- ▶ We'll consider composition of functors many times in this course.
- ▶ We'll sometimes simplify notation, writing  $FGa$ ,  $FGHa$ , etc. (Why)'s this ok?

## Demo at [schar.github.io/TDParse](https://schar.github.io/TDParse)

```
> ann saw her  
> she saw ann  
> ann said she saw her mom  
> someone saw everyone  
> ...
```

## Comments, questions, concerns addressed

Are occurrences of • actually instantiated in the syntax?

- ▶ **No!** • is in the **interpreter**, not the language

How is this *practical*? LIFT and • can apply iteratively.

- ▶ Combination is recursive, but this recursion is pointed **down** the type hierarchy, rather than up. Semantic parsing is decidable!

The system is unwieldy. Derivations are complex and difficult to construct.

- ▶ Our semantic parses are **homomorphic** to the syntax that generates them. Construction of derivations is **automatic** (but not complicated for humans).

What about PM, etc? How does • help us with, e.g., *dog near her*?

- ▶ Let's check...

Under the hood

## Where do meanings come from?

So far, we've seen that Haskell is a pretty apt **metalanguage** for natural language. But we haven't seen how these meanings could be generated in the first place.

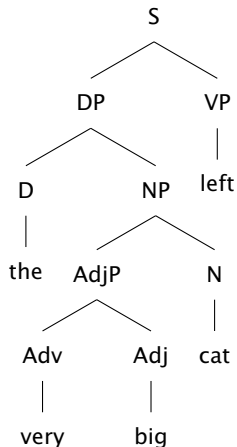
When writing Haskell, the compiler knows, in a type-directed way, which • is intended, but it will not insert it (or LIFT's) for you.

These are problems of structuring unstructured input, i.e., parsing. We cannot rely on Haskell's parsing and type inference facilities, but we can build on them.

## Representing syntactic objects (constituency only)

```
data Syn = Leaf String
         | Branch Syn Syn

s1 :: Syn
s1 = Branch -- S
      (Branch -- DP
        (Leaf "the") -- D
        (Branch -- NP
          (Branch -- AdjP
            (Leaf "very") -- Adv
            (Leaf "big")) -- Adj
          (Leaf "cat")) -- N
        (Leaf "left")) -- VP
```



## Encoding syntax and semantics

**data** Syn

= Leaf String  
| Branch Syn Syn

**data** Sem

= Lex String  
| Comp Mode Sem Sem

**data** Mode

= FA | BA  
| PM -- *etc*

**data** Type

= E | T  
| Type :-> Type

## Implementing type-driven combination

$$\llbracket A B \rrbracket ::= \begin{cases} \llbracket A \rrbracket \llbracket B \rrbracket & \text{if } A : \sigma \rightarrow \tau, B : \sigma & \text{FA} \\ \llbracket B \rrbracket \llbracket A \rrbracket & \text{if } A : \sigma, B : \sigma \rightarrow \tau & \text{BA} \\ \llbracket A \rrbracket \cap \llbracket B \rrbracket & \text{if } A, B : \sigma \rightarrow \tau & \text{PM} \\ \dots & \dots & \dots \end{cases}$$

```
combine :: Type -> Type -> [(Mode, Type)]
combine l r =
  [(FA, b) | a :-> b <- [l], a == r] ++
  [(BA, b) | a :-> b <- [r], a == l] ++
  [(PM, a :-> T) | a :-> T <- [l], b :-> T <- [r], a == b]
  -- ...
```

## Examples of type-driven composition

```
ghci> combine E (E :-> T)
[(BA, T)]
```

```
ghci> combine (E :-> T) (E :-> T)
[(PM, E :-> T)]
```

```
ghci> combine (E :-> T) T
[]
```

## Syntax to semantics

The following implements the standard logic of recursive type-driven composition:

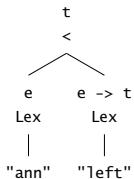
```
synsem :: Syn -> [(Sem, Type)]
synsem (Leaf w) =
  [(Lex w, ty) | ty <- lex w]
synsem (Branch l r) =
  [(Comp op lsem rsem, ty) | (lsem, lty) <- synsem l
                             , (rsem, rty) <- synsem r
                             , (op, ty) <- combine lty rty]
```

```
ghci> s0 = Branch (Leaf "ann") (Leaf "left")
ghci> synsem s0
(Comp BA (Lex "ann") (Lex "left"), T)

ghci> synsem s1
((Comp BA (Comp FA (Lex "the") (Comp PM (Comp FA (Lex "very")
  (Lex "big"))) (Lex "cat")) (Lex "left")), T)
```

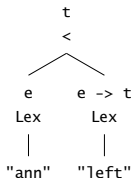
## Semantic values as (syntax-homomorphic) trees

```
ghci> semTrees s0
```

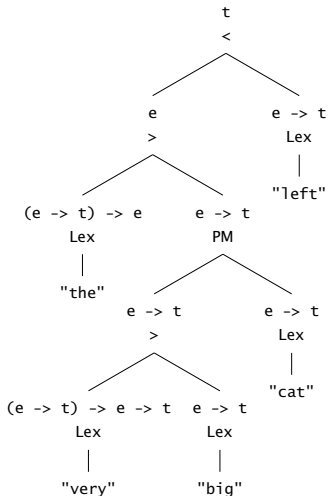


## Semantic values as (syntax-homomorphic) trees

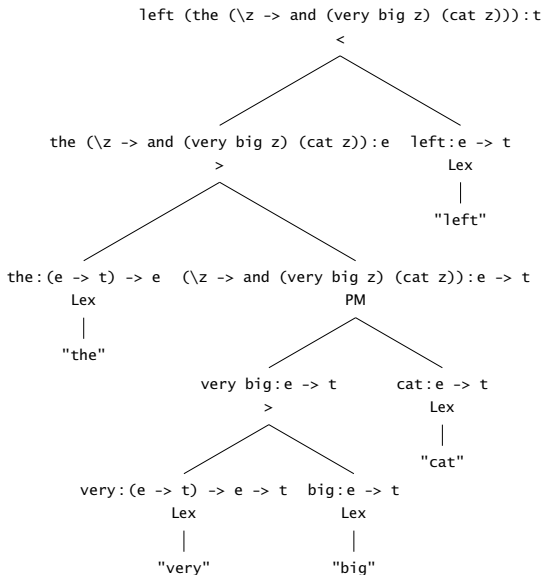
```
ghci> semTrees s0
```



```
ghci> semTrees s1
```



## With some term normalization



## Adding effectful things to the grammar

Some notions of computational effects to get us going

```
data F = R | S | W | C -- ...
```

```
functor :: F -> Bool  
functor _ = True
```

Regular types extended with computational types:

```
data Type = E | T  
          | Type :-> Type  
          | Comp F Type
```

## Meta-modes

```
data Mode
  = FA | BA | PM          -- Base modes          >, <, &
  | MR Mode | ML Mode    -- Meta-modes for functors  fmap
```

Then extending our type-driven interpreter just amounts to extending `combine!`

## Extending combine via functorial meta-modes

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{(\lambda l'. l' * r) \bullet l : Fc}{l : Fa \quad r : b} \bar{F}*$$

$$\text{If } \frac{l * r : c}{l : a \quad r : b} * , \text{ then } \frac{(\lambda r'. l * r') \bullet r : Fc}{l : a \quad r : Fb} \bar{F}*$$

```
combine :: Type -> Type -> [(Mode, Type)]
```

```
combine l r = oldCombine l r
```

```
++ [(ML op, Comp f c) | Comp f a <- [],  
                        , functor f  
                        , (op, c) <- combine a r]
```

```
++ [(MR op, Comp f c) | Comp f b <- [r],  
                        , functor f  
                        , (op, c) <- combine l b]
```

## Some combinations

```
ghci> combine (Comp R E) (E :-> T)
[(ML BA, Comp R T)]
```

```
ghci> combine E (Comp R (E :-> T))
[(MR BA, Comp R T)]
```

```
ghci> combine (Comp R E) (Comp R (E :-> T))
[(ML (MR BA), Comp R (Comp R T)),
 (MR (ML BA), Comp R (Comp R T))]
```

```
ghci> combine (Comp S E) (Comp R (E :-> T))
[(ML (MR BA), Comp S (Comp R T)),
 (MR (ML BA), Comp R (Comp S T))]
```

- Barker, Chris. 2002. Continuations and the nature of quantification. *Natural Language Semantics* 10(3). 211–242. <https://doi.org/10.1023/A:1022183511876>.
- Beaver, David & Emiel Krahmer. 2001. A partial account of presupposition projection. *Journal of logic, language and information* 10(2). 147–182.
- Boër, Steven E & William G Lycan. 1976. The myth of semantic presupposition.
- Bumford, Dylan & Simon Charlow. 2025. Effect-driven interpretation: Functors for natural language composition. In press, Cambridge University Press, *Elements in Semantics*.
- Charlow, Simon. 2014. *On the semantics of exceptional scope*. New York University Ph.D. thesis. <https://semanticsarchive.net/Archive/2JmMWRjY/>.
- Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2). 117–184. <https://doi.org/10.1023/A:1005464228727>.
- Kratzer, Angelika & Junko Shimoyama. 2002. Indeterminate pronouns: The view from Japanese. In Yukio Otsu (ed.), *Proceedings of the Third Tokyo Conference on Psycholinguistics*, 1–25. Tokyo: Hituzi Syobo.