Effectful composition in natural language semantics

Multiple functors, automating composition

Dylan Bumford (UCLA) Simon Charlow (Rutgers)

ESSLLI 2022, NUI Galway

Recap, and a little category theory

Denotations via functors

Expression	Type	Denotation
no cat	$\mathtt{Ce} ::= (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	$\lambda c.\neg\exists x.\mathbf{cat}\,x \wedge c\,x$
the cat	$\mathtt{Me} ::= \mathtt{e} \mid \#$	x if $\mathbf{cat} = \{x\}$ else #
Sassy, a cat	$\mathtt{We} ::= \mathtt{e} \times \mathtt{t}$	$\langle \mathbf{s}, \mathbf{cat}\,\mathbf{s}\rangle$
she	$\mathtt{Re} ::= \mathtt{r} \to \mathtt{e}$	$\lambda g.g_0$
which cat	$\mathtt{Se} ::= \{\mathtt{e}\}$	$\{x \mid \mathbf{cat}\,x\}$
SASSY	$\mathtt{Fe} ::= \mathtt{e} \times \{\mathtt{e}\}$	$\langle \mathbf{s}, \{x \mid x \in D_e\}\rangle$
a cat	$\mathtt{De} ::= \mathtt{s} \to \{\mathtt{e} \times \mathtt{s}\}$	$\lambda s.\{\langle x, s +\!\!+ x\rangle \mid \mathbf{cat}\,x\}$
...	...	...

It is worth contemplating the hoops you'd need to jump through to develop a theory of grammar in the standard mold that could handle all these effects (and more).

- The theory would be incredibly compex, inflexible, and brittle.

Recap

A functor is a type constructor (read: **notion of fanciness**) that supports an `fmap`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
-- fmap id == id
-- fmap (f . g) == fmap f . fmap g
```

The second of these laws can't help but be satisfied. The first guarantees a kind of "well-behavedness" (e.g., `fmap f xs = []` doesn't qualify).

Haskell will ensure that any `Functor` instances you declare are appropriately typed, but it can't prove that `fmap id == id`. That is your job.

Concrete example 1: mapping over lists

`fmap`-ing a function `f :: a -> b` over a list `xs :: [a]` yields a new list of type `[b]`, formed by applying `f` to each element of `xs`:

```
instance Functor [] where -- [] is the type constructor for lists
  fmap f xs = [f x | x <- xs]
```

(In Haskell, `fmap` for lists is also known simply as `map`.)

A couple examples of how this works:

```
fmap succ [1, 2, 3]
  == [succ 1, succ 2, succ 3]
  == [2, 3, 4]
```

```
fmap even [1, 2, 3]
  == [even 1, even 2, even 3]
  == [False, True, False]
```

The functor laws for `fmap f xs = [f x | x <- xs]`

```
-- fmap id == id
fmap id xs
  == [id x | x <- xs] -- def of fmap
  == [x | x <- xs]    -- def of id
  == xs               -- ==
```

The functor laws for `fmap f xs = [f x | x <- xs]`

```
-- fmap id == id
fmap id xs
  == [id x | x <- xs] -- def of fmap
  == [x | x <- xs]    -- def of id
  == xs               -- ==

-- fmap (f . g) == fmap f . fmap g
fmap (f . g) xs
  == [(f . g) x | x <- xs]  -- def of fmap
  == [f (g x) | x <- xs]    -- def of (.)
  == fmap f [g x | x <- xs] -- def of fmap
  == fmap f (fmap g xs)     -- def of fmap
  == (fmap f . fmap g) xs   -- def of (.)
```

Concrete example 2: mapping over context-sensitive meanings

`fmap`-ing `f :: a -> b` over `rx :: r -> a` yields a context-sensitive `r -> b` by reading in some `r`, using it to get a value out of `rx`, and applying `f` to the result:

```
instance Functor ((->) r) where -- (->) r is what we're calling R
  fmap f rx = f . rx
  --        == \r -> f (rx r)
```

A couple examples of how this works, assuming `var2 r = r!!2`:

```
fmap succ var2
  == succ . var2
  == \r -> succ (var2 r)
  == \r -> succ (r!!2)
```

```
fmap even var2
  == even . var2
  == \r -> even (var2 r)
  == \r -> even (r!!2)
```

Suppose we pass in `[7,8,9]` as `r`. What do we get?

Concrete example 2: mapping over context-sensitive meanings

`fmap`-ing `f :: a -> b` over `rx :: r -> a` yields a context-sensitive `r -> b` by reading in some `r`, using it to get a value out of `rx`, and applying `f` to the result:

```
instance Functor ((->) r) where -- (->) r is what we're calling R
  fmap f rx = f . rx
  --        == \r -> f (rx r)
```

A couple examples of how this works, assuming `var2 r = r!!2`:

```
fmap succ var2
  == succ . var2
  == \r -> succ (var2 r)
  == \r -> succ (r!!2)
```

```
fmap even var2
  == even . var2
  == \r -> even (var2 r)
  == \r -> even (r!!2)
```

Suppose we pass in `[7,8,9]` as `r`. What do we get? `10`, `False`.

The functor laws for `fmap f rx = f . rx`

```
-- fmap id == id
fmap id rx
  == id . rx -- def of fmap
  == rx      -- id is identity element for (.)
```

The functor laws for `fmap f rx = f . rx`

```
-- fmap id == id
fmap id rx
  == id . rx -- def of fmap
  == rx      -- id is identity element for (.)

-- fmap (f . g) == fmap f . fmap g
fmap (f . g) rx
  == (f . g) . rx          -- def of fmap
  == f . (g . rx)          -- (.) is associative
  == fmap f (g . rx)       -- def of fmap
  == fmap f (fmap g rx)    -- def of fmap
  == (fmap f . fmap g) rx -- def of (.)
```

Categories

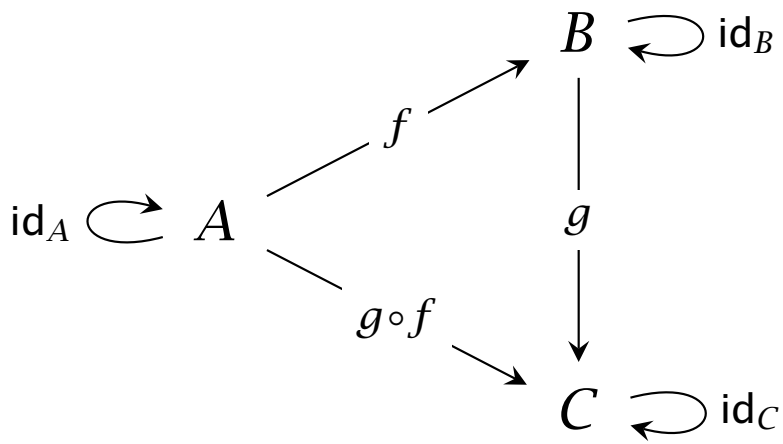
A **category** is a database of stuff, related in a compositional way:

- A collection of **objects** $(A, B, C, \ldots)$ — stuff
- A collection of **morphisms** between objects $(f : A \to B, \ldots)$ — related
- A **composition** operation $\circ$ such that for any $f : A \to B$ and $g : B \to C$, there is a morphism $g \circ f : A \to C$ (pronounced, 'g after f') — compositionally

Composition should be well-behaved in the following sense:

- Associativity: $(f \circ g) \circ h = f \circ (g \circ h)$ (neutrally: '$f \circ g \circ h$')
- For each object x, there's an an identity morphism $\mathsf{id}_X : X \to X$, such that given $f : A \to B$, $f \circ \mathsf{id}_A = f$, and $\mathsf{id}_B \circ f = f$.

Diagrammatically



Examples of categories

The category Set...

- Objects are sets; morphisms are set-theoretic functions between sets
- $\circ$ is function composition, $(f \circ g)\, x = f\,(g\, x)$

The category Hask...[1]

- Objects are Haskell types; morphisms are typed Haskell functions
- $\circ$ is composition of Haskell functions, `(f . g) x = f (g x)`

Any preorder $(S, \leqslant)$...

- Objects are the elements of S; $f : a \to b$ iff $a \leqslant b$
- Reflexivity and transitivity of $\leqslant$ make a category

[1] Not *really* (non-termination complicates things), but close enough to be illuminating, useful.

Functors

A functor is a type constructor with an `fmap`:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b -- fmap id == id
                                 -- fmap (f . g) == fmap f . fmap g
```

In terms of categories, a **functor** F is a mapping between categories C and $\mathcal{D}$...

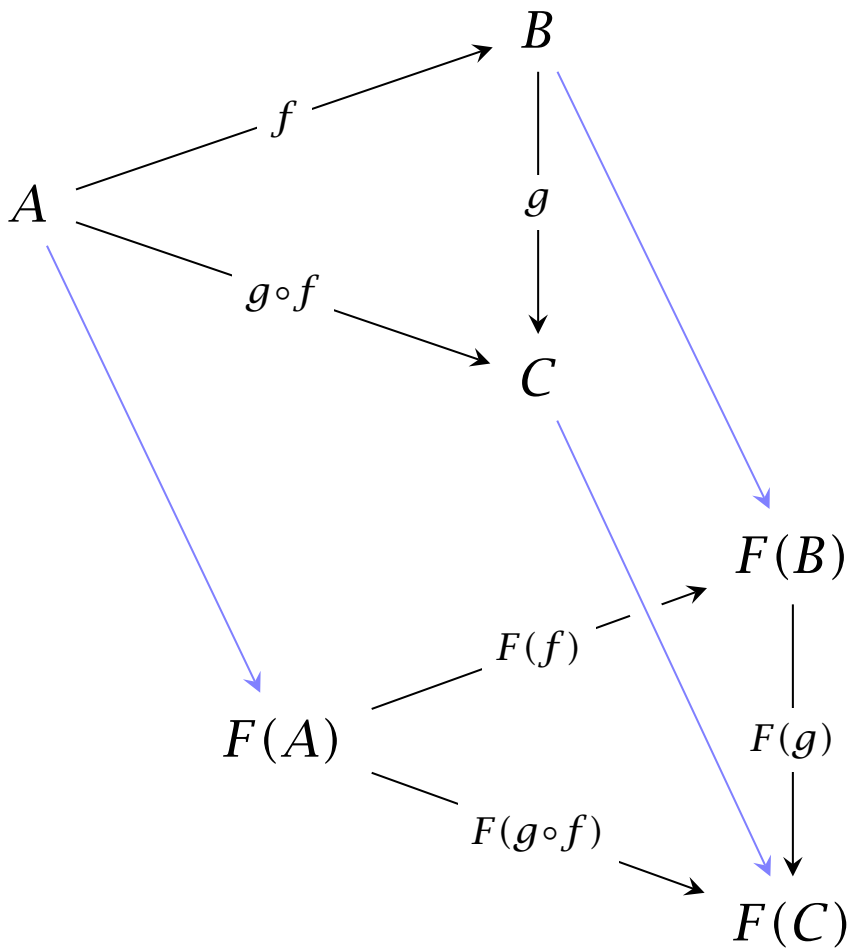
- F associates each a in C with $F(a)$ in $\mathcal{D}$ (cf. type constructor)
- F associates each $f : a \to b$ in C with $F(f) : F(a) \to F(b)$ in $\mathcal{D}$ (cf. `fmap`)

...Which preserves the compositional structure of C:

- $F(\mathsf{id}_a) = \mathsf{id}_{F(a)}$
- $F(f \circ g) = F(f) \circ F(g)$ (cf. `fmap` laws)

An **endofunctor** is a functor from C to C (e.g., from Hask to Hask).

Diagrammatically



Why categories?

While we will not foreground category-theoretic considerations in this course, **composition** and **compositionality** will be central concerns.

From this vantage, we see that the essence of functors is **transferable skills**:

- Having a functor F means: whatever I know how to do in some "lower" space, I know how to do in the "higher" space characterized by F.

Why categories?

While we will not foreground category-theoretic considerations in this course, **composition** and **compositionality** will be central concerns.

From this vantage, we see that the essence of functors is **transferable skills**:

- Having a functor F means: whatever I know how to do in some "lower" space, I know how to do in the "higher" space characterized by F.

We'd invite you to think of the diagram on the previous slide as a (very skeletal) schematization of this grammatical "lifting":

- Old grammar + functorial F = new grammar that seamlessly deals with F
- Toss a functor in, shake, and reap the rewards.

Could it really be so simple?

Why categories?

While we will not foreground category-theoretic considerations in this course, **composition** and **compositionality** will be central concerns.

From this vantage, we see that the essence of functors is **transferable skills**:

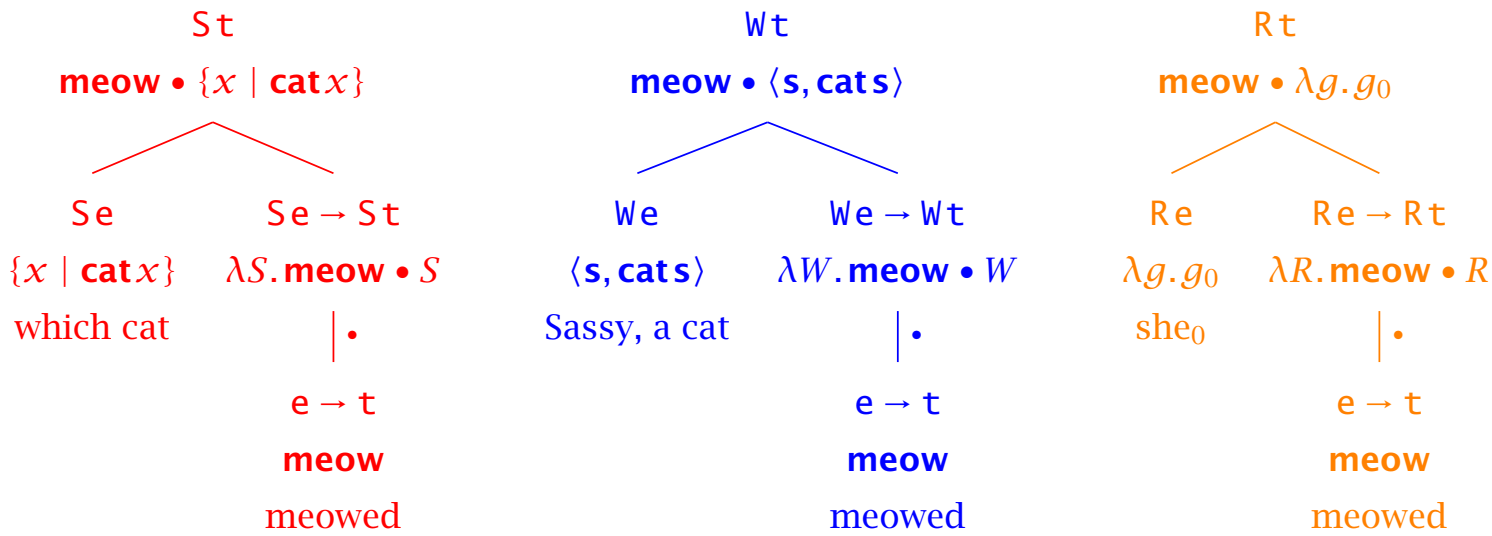
- Having a functor F means: whatever I know how to do in some "lower" space, I know how to do in the "higher" space characterized by F.

We'd invite you to think of the diagram on the previous slide as a (very skeletal) schematization of this grammatical "lifting":

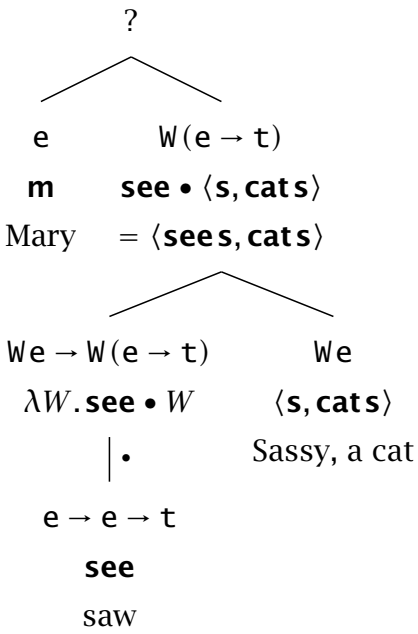
- Old grammar + functorial F = new grammar that seamlessly deals with F
- Toss a functor in, shake, and reap the rewards.

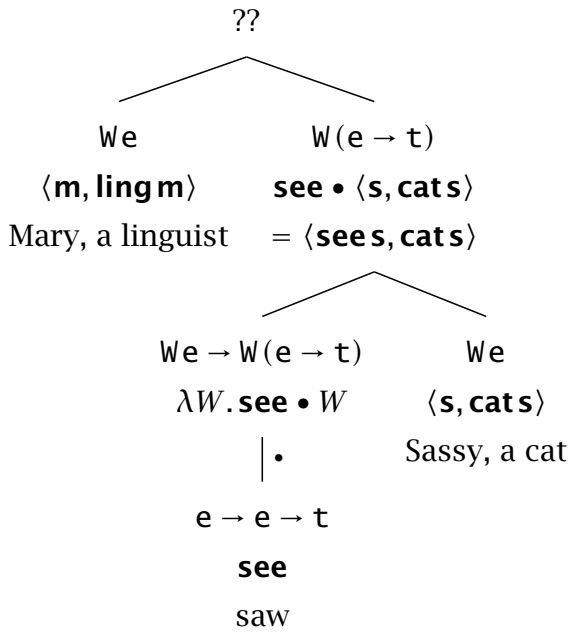
Could it really be so simple? Actually, yeah, if you line things up in the right way.

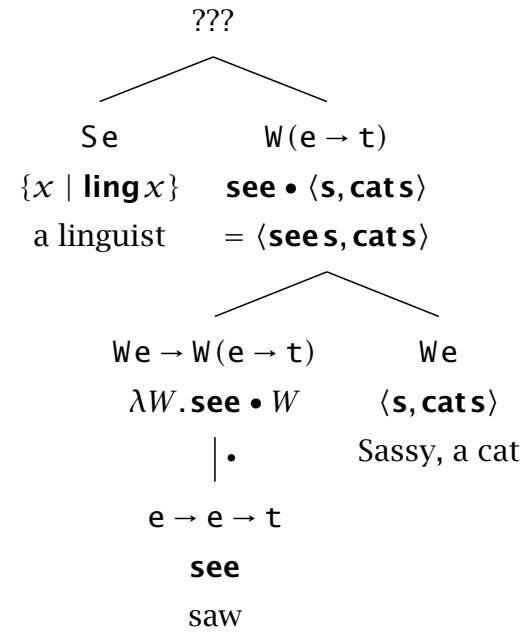
Composition with functors: some immediate successes



Some more puzzling configurations







Functorial case study: variable-free semantics

The ‘standard’ picture

$$
[\![A\ B]\!] ::= \begin{cases}
[\![A]\!][\![B]\!] & \text{if } A :: \sigma \to \tau,\ B :: \sigma & \text{FA} \\
[\![B]\!][\![A]\!] & \text{if } A :: \sigma,\ B :: \sigma \to \tau & \text{BA} \\
[\![A]\!] \cap [\![B]\!] & \text{if } A, B :: \sigma \to \mathsf{t} & \text{PM} \\
[\![A]\!] \circ [\![B]\!] & \text{if } A :: \tau \to \upsilon,\ B :: \sigma \to \tau & \text{FC} \\
[\![B]\!] \upharpoonright [\![A]\!] & \text{if } A :: \sigma \to \mathsf{t},\ B :: \sigma \to \tau \to \mathsf{t} & \text{PR} \\
\ldots & \text{if } \ldots & \ldots
\end{cases}
$$

Environment-dependence

Natural languages have free and bound pro-forms.

1. John saw her. I wouldn't _ if I were you.
2. Everybody$_i$ did their$_i$ homework. When I'm supposed to work$_i$ I can't _$_i$.

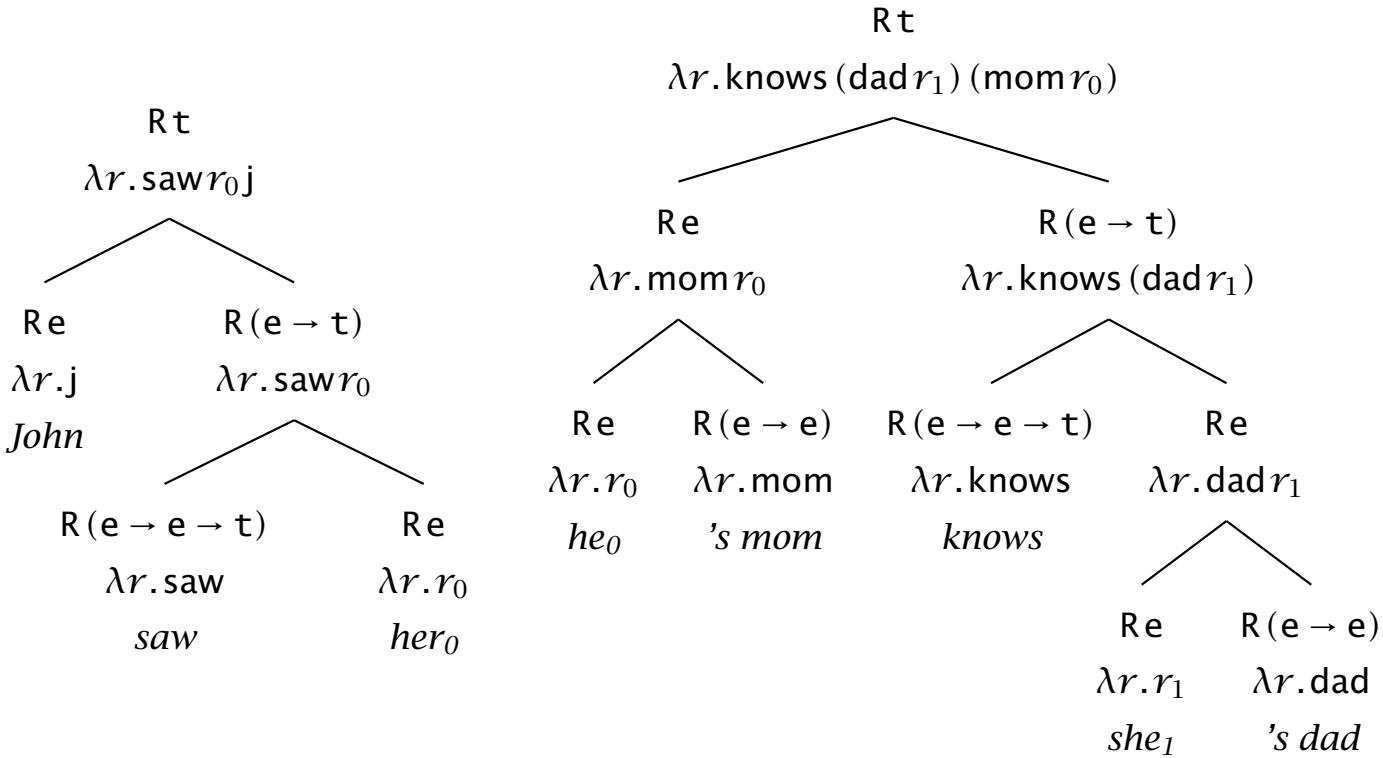
Standardly: meanings depend on environments (ways of valuing free variables).

$$\sigma ::= \mathtt{e} \mid \mathtt{t} \mid \sigma \to \sigma \qquad\qquad \tau ::= \mathsf{R}\,\sigma ::= \mathtt{r} \to \sigma$$

Interpret binary combination via **environment-sensitive** functional application.

$$[\![\alpha\ \beta]\!] := \lambda r.\underbrace{\underbrace{[\![\alpha]\!]}_{\mathsf{R}(b\to c)}\, r\ (\underbrace{[\![\beta]\!]}_{\mathsf{R}b}\, r)}_{\mathsf{R}c}$$

A couple examples



Apply the result to a salient environment, i.e., assignment

Pronouns as identity maps

Jacobson (1999) says not to treat pronouns as indexed and assignment-relative.

Instead: model pronouns as **identity functions**:

$$\mathsf{R}a ::= \mathsf{e} \to a \qquad\qquad \mathsf{she} := \underbrace{\lambda x.x}_{\mathsf{Re}}$$

How should these compose with things like transitive verbs, which are looking for an individual, not a function from individuals to individuals?

Pronouns as identity maps

Jacobson (1999) says not to treat pronouns as indexed and assignment-relative. Instead: model pronouns as **identity functions**:

$$\mathsf{R}a ::= \mathsf{e} \to a \qquad\qquad \mathsf{she} := \underbrace{\lambda x.x}_{\mathsf{Re}}$$

How should these compose with things like transitive verbs, which are looking for an individual, not a function from individuals to individuals?

The tension here is similar to what Dylan discussed yesterday. Expressions may behave as if they have type e, but carry aspects of meanings too 'big' to fit inside e.

$$\mathsf{e} \not\equiv \mathsf{Re}$$

Heim & Kratzer (1998: 92): $[\![t]\!] = \lambda x . x$

(5) Preliminary definition: An *assignment* is an individual (that is, an element of D (= D_e)).

A trace under a given assignment denotes the individual that constitutes that assignment; for example:

(6) The denotation of "t" under the assignment Texas is Texas.

An appropriate notation to abbreviate such statements needs to be a little more elaborate than the simple [[. . .]] brackets we have used up to now. We will indicate the assignment as a superscript on the brackets; for instance, (7) will abbreviate (6):

(7) $[\![t]\!]^{\text{Texas}}$ = Texas.

The general convention for reading this notation is as follows: Read "$[\![\alpha]\!]^a$" as "the denotation of α under a" (where α is a tree and a is an assignment).

(7) exemplifies a special case of a general rule for the interpretation of traces, which we can formulate as follows:

(8) If α is a trace, then, for any assignment a, $[\![\alpha]\!]^a$ = a.

Three approaches to composition

Environment-insensitive composition:

$$\frac{f\,x : b}{f : a \to b \qquad x : a}$$

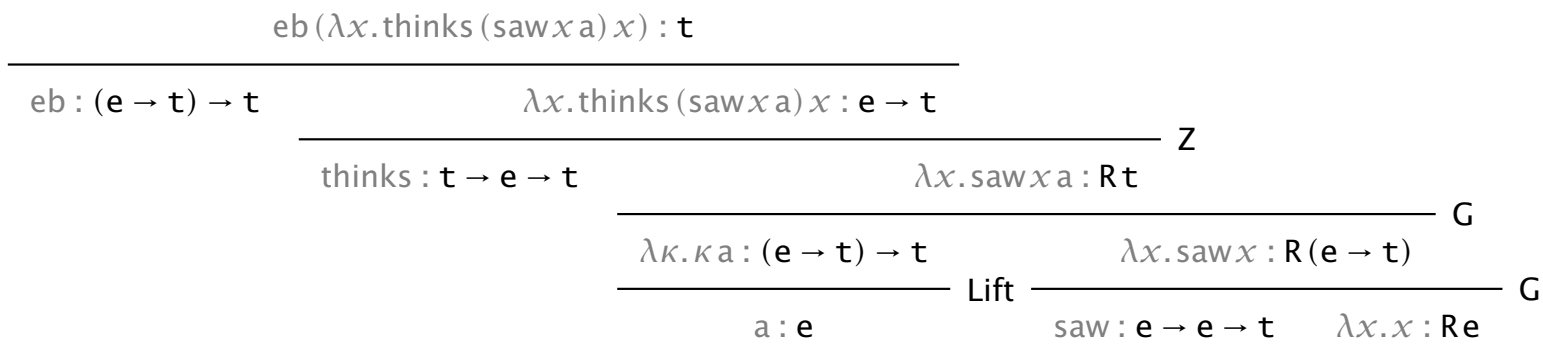
Environment-sensitive composition everywhere:

$$\frac{\lambda g.\, f\, g\, (x\, g) : \mathsf{R}\, b}{f : \mathsf{R}\,(a \to b) \qquad x : \mathsf{R}\, a}$$

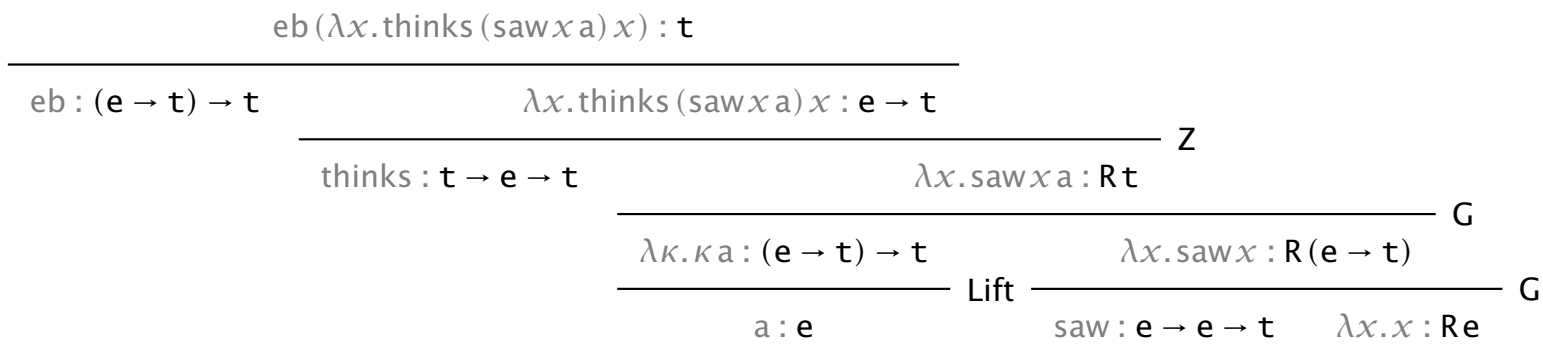
Jacobson's approach: Environment-sensitive composition, on demand:

$$\frac{\lambda x.\, f\, (m\, x) : \mathsf{R}\, b}{f : a \to b \qquad m : \mathsf{R}\, a}\ \mathsf{G} \qquad \frac{\lambda x.\, f\, (m\, x)\, x : \mathsf{e} \to b}{f : a \to \mathsf{e} \to b \qquad m : \mathsf{R}\, a}\ \mathsf{Z}$$

VFS derivation



VFS derivation



Despite differences of presentation, we can clearly see that G is none other than R's `fmap` (though presented here as a binary operation).

Multiple pronouns

Assignments are data structures that can in principle value every free pronoun you need. But an individual can only value co-valued pronouns!

3. She saw her.

So a variable-free treatment of cases like these must give something like this:

$$\underbrace{\lambda x.\lambda y.\mathsf{saw}\, y\, x}_{\mathtt{R\,(R\,t)}}$$

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

Functors compose

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

$$(\bullet)\,((\bullet)\,f) :: F\,(G\,a) \to F\,(G\,b)$$

$$\Big|\,\bullet$$

$$(\bullet)\,f :: G\,a \to G\,b$$

$$\Big|\,\bullet$$

$$f :: a \to b$$

```
Prelude> :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 $FG\,a$, $FGH\,a$, etc. (Why)'s this ok?

Ross Paterson's `Data.Functor.Compose` (on Hackage)

```haskell
module Data.Functor.Compose (
    Compose(..),
  ) where

newtype Compose f g a = Compose { getCompose :: f (g a) }

instance (Functor f, Functor g) => Functor (Compose f g) where
  fmap f (Compose x) = Compose (fmap (fmap f) x)
```

`fmap fmap`

Composition of functors alone will not be enough to generate every meaning we need. To derive sentences with multiple pronouns, we must also `fmap fmap`!

$$(\bullet)\,(\bullet) :: F\,(a \to b) \to F\,(G\,a \to G\,b)$$

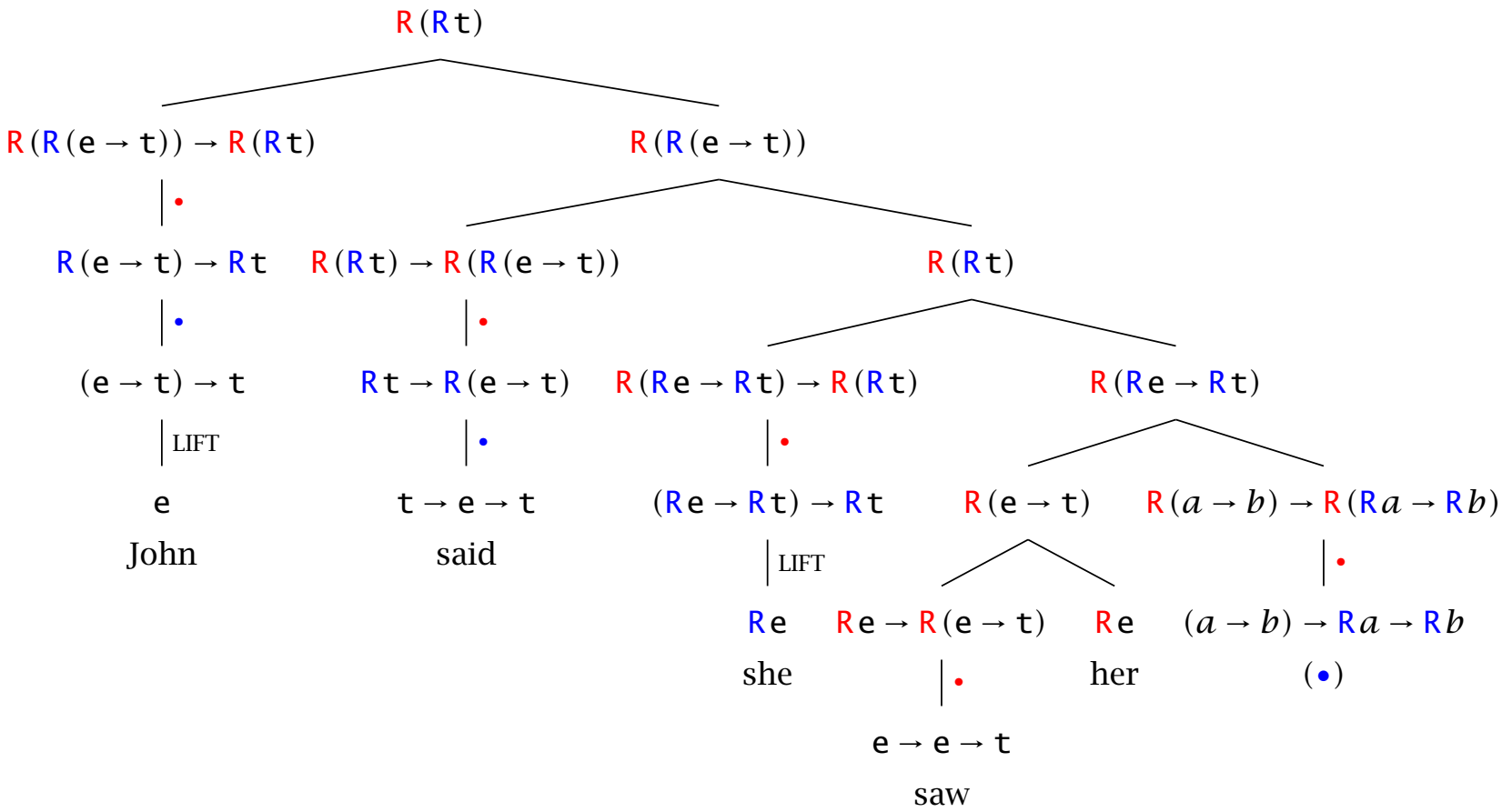
$$\Big|\ \bullet$$

$$(\bullet) :: (a \to b) \to G\,a \to G\,b$$

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

Nullary occurrences of our functorial combinators raise difficult questions about the syntax and the relationship between syntax and semantics.

Hardcore VFS derivation: *John said she saw her*



Mfw spending an hour on the last slide

Mfw spending an hour on the last slide

We're a long way from homomorphic...

Comments, questions, concerns

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

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

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

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

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

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

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

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

What about PM, etc? How does $(\bullet)$ help us with, e.g., *dog near her*?

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

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

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

What about PM, etc? How does $(\bullet)$ help us with, e.g., *dog near her*?

Must we always be committed to higher-order meanings whenever we are in the presence of multiple effectful things? Could get ugly...

Comments, questions, concerns

Are occurrences of $(\bullet)$ actually instantiated in the syntax??

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

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

What about PM, etc? How does $(\bullet)$ help us with, e.g., *dog near her*?

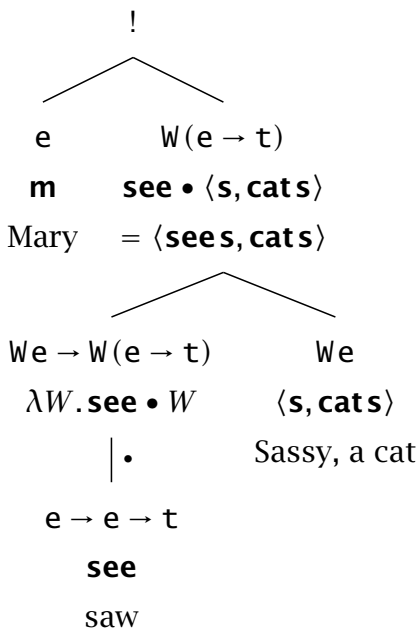
Must we always be committed to higher-order meanings whenever we are in the presence of multiple effectful things? Could get ugly...

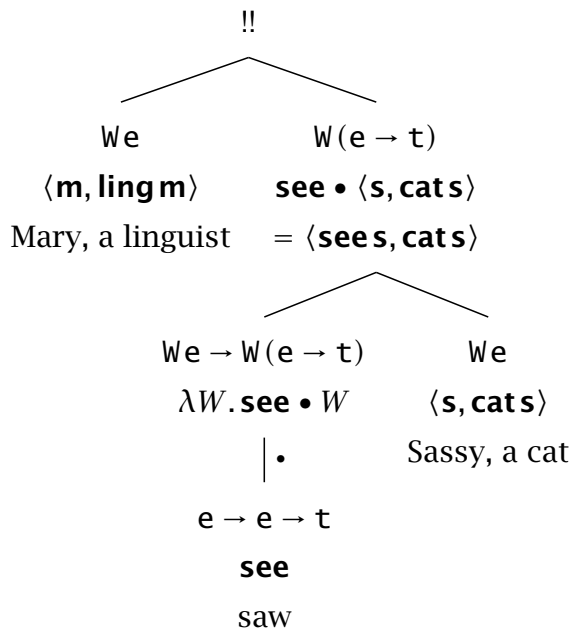
Empirical adequacy: binding does not require c-command!

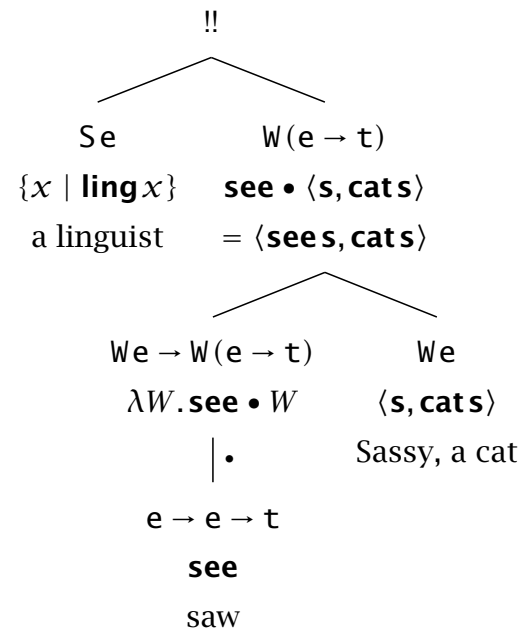
Slightly less puzzled

We can improve on all these fronts, and we will.

But *empirically* we are in a reasonable place: our functors enable us to ignore the extra structure in which our values of compositional interest are embedded.



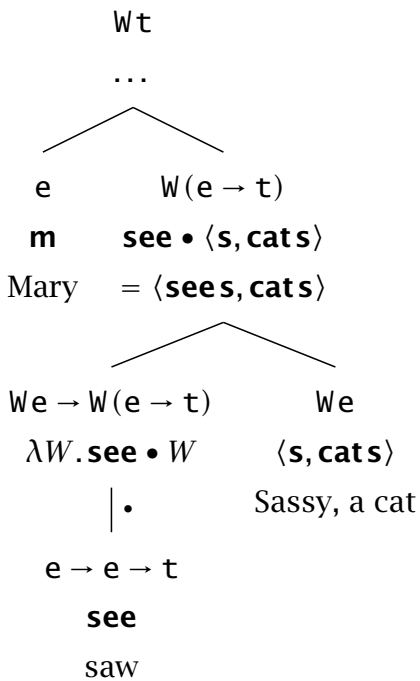


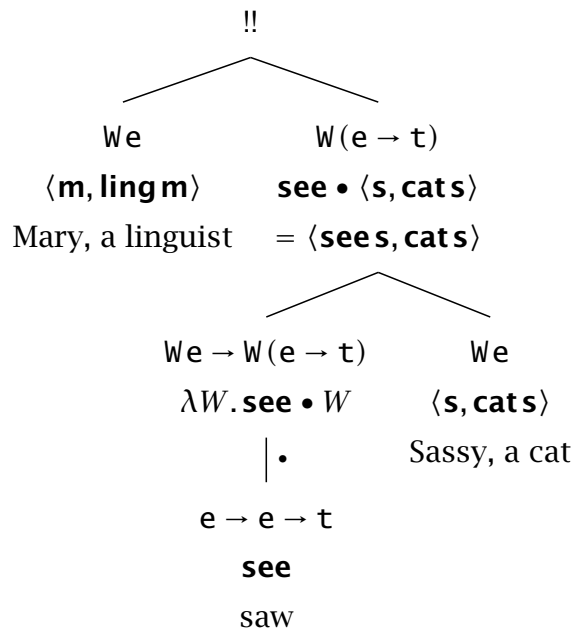


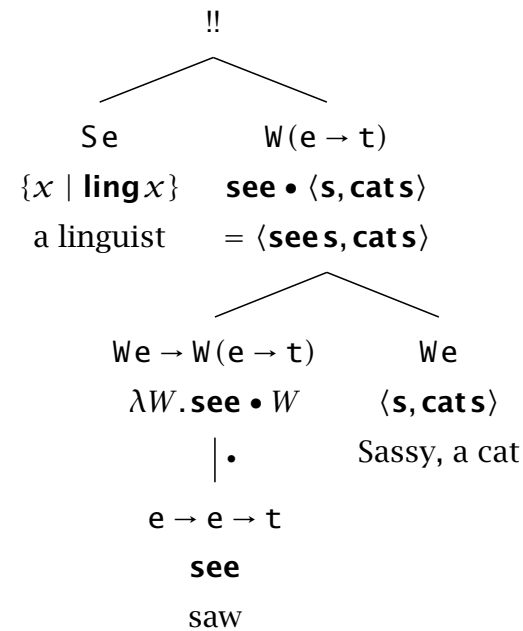
Slightly less puzzled

We can improve on all these fronts, and we will.

But *empirically* we are in a reasonable place: our functors enable us to ignore the extra structure in which our values of compositional interest are embedded.



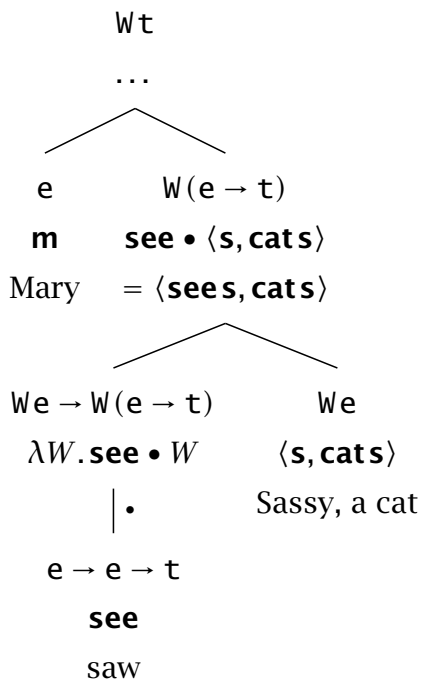


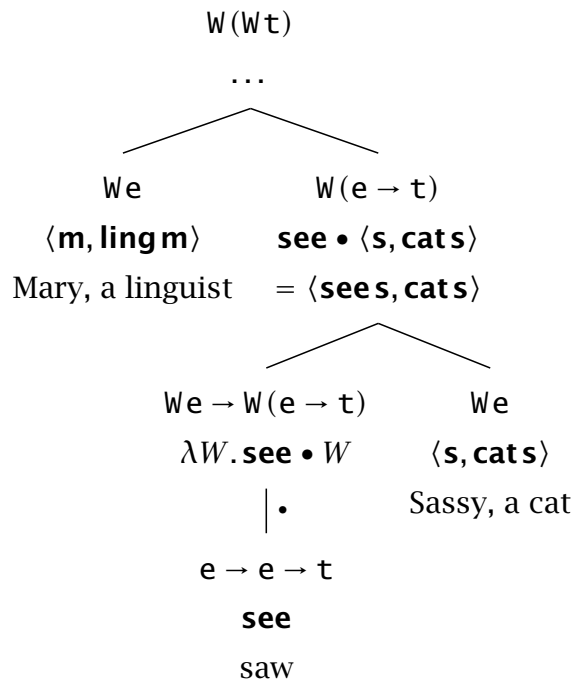


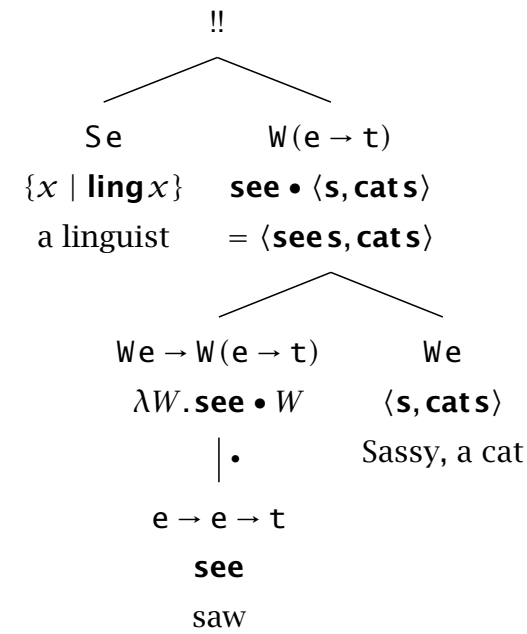
Slightly less puzzled

We can improve on all these fronts, and we will.

But *empirically* we are in a reasonable place: our functors enable us to ignore the extra structure in which our values of compositional interest are embedded.



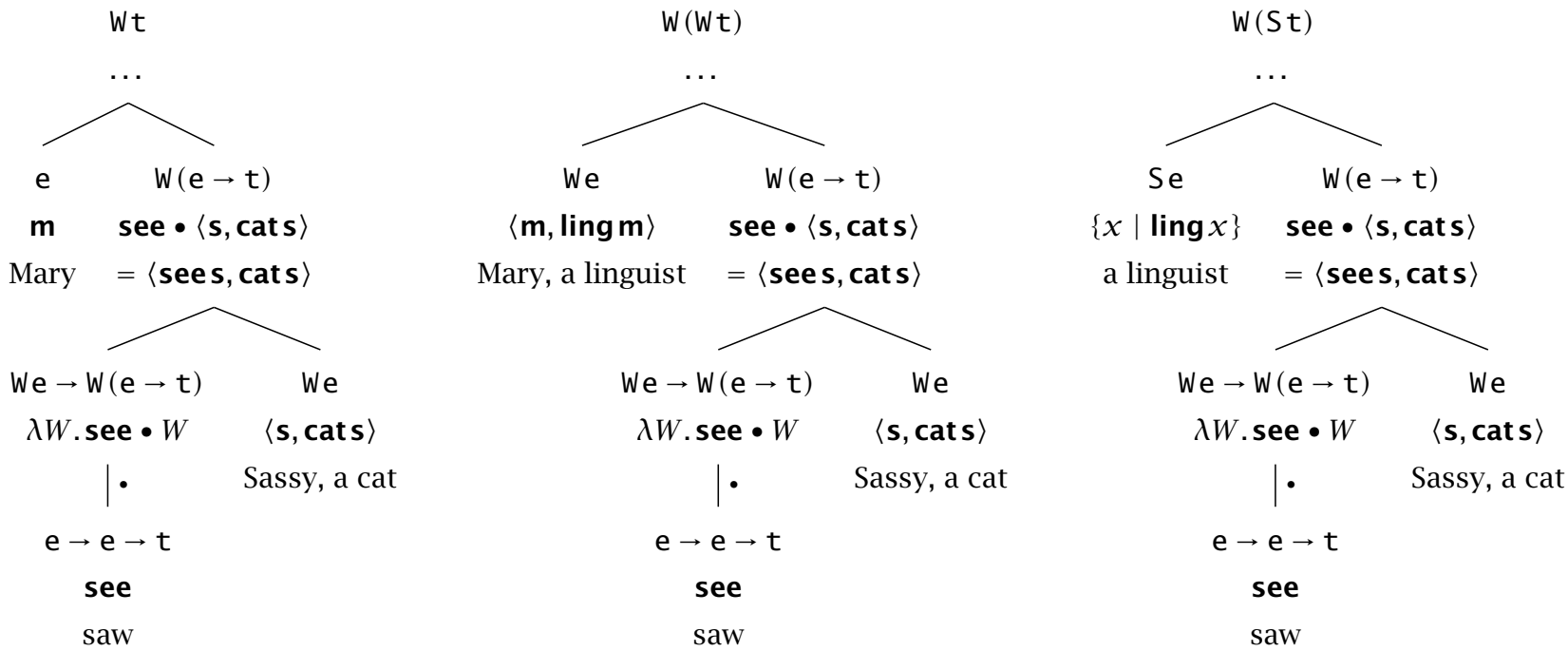




Slightly less puzzled

We can improve on all these fronts, and we will.

But *empirically* we are in a reasonable place: our functors enable us to ignore the extra structure in which our values of compositional interest are embedded.



The variable-full semanticist's POV

The variable-free semanticist has a compelling case for higher-order structure. Does the variable-full semanticist?

We think so. Not to handle multiple pronouns. An assignment is big enough for that already. But possibly to value variables of different types:[2]

4. And [buy the couch]$_1$ $\underbrace{\text{she}_2 \text{ did } \text{t}_1}_{\lambda g.\lambda h.\, g_1\, h_2}$.

Yet it seems likely that variable-full semanticists would also wish to derive a regular, non-higher-order meaning for *she*$_0$ *saw her*$_1$.

- Functors and `fmap` alone can never do that.
- But other, related constructs will get the job done. Stick around.

[2] I've argued (Charlow 2019) that this yields a natural account of paycheck pronouns.

Type-driven (effectful) composition in Haskell

Some disanalogies between programming lgs and natural lgs

Unlike Haskell syntax, natural language syntax is **ambiguous**:

5. I saw the kestrel with the binoculars.

Even fixing a syntactic structure, ambiguity remains:

6. A doctor examined every patient.
7. Put a chicken in every pot.
8. Send a message to every VP.

And sometimes disappears:

9. I know a doctor who examined every patient.
10. A doctor examined every patient. She was quite busy.

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.

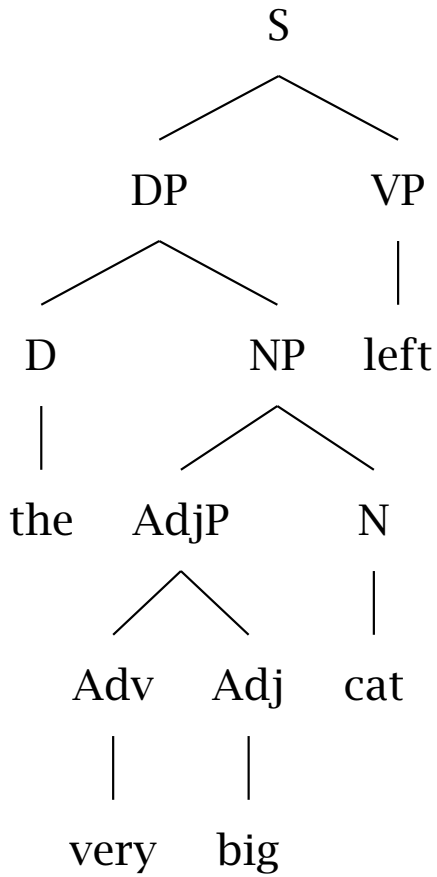
When writing Haskell, the compiler knows, in a type-directed way, which • is intended, but it will not insert it (or LIFT's) for you.

These issues (and those on the previous slide) are all problems of structuring unstructured input, i.e., parsing. We cannot entirely rely on Haskell's parsing and type inference facilities, but we can build on them.

Representing syntactic objects (constituency only)

```haskell
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
```

Type-driven combination

$$[\![A\ B]\!] ::= \begin{cases} [\![A]\!][\![B]\!] & \text{if } A :: \sigma \to \tau,\ B :: \sigma \quad \text{FA} \\ [\![B]\!][\![A]\!] & \text{if } A :: \sigma,\ B :: \sigma \to \tau \quad \text{BA} \\ [\![A]\!] \cap [\![B]\!] & \text{if } A, B :: \sigma \to t \quad \text{PM} \\ \ldots & \text{if } \ldots \quad \ldots \end{cases}$$

```
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

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

*TypeDriven> combine (E :-> T) (E :-> T)
[(PM, E :-> T)]

*TypeDriven> 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 lval rval, ty) | (lval, lty) <- synsem l
                            , (rval, rty) <- synsem r
                            , (op, ty) <- combine lty rty ]
```

Syntax to semantics

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

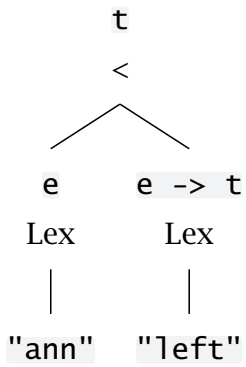
```
synsem :: Syn -> [(Sem, Type)]
synsem (Leaf w) = [(Lex w, ty) | ty <- lex w]
synsem (Branch l r) =
  [ (Comp op lval rval, ty) | (lval, lty) <- synsem l
                            , (rval, rty) <- synsem r
                            , (op, ty) <- combine lty rty ]

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

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

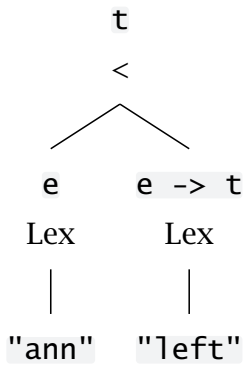
Semantic values as (syntax-homomorphic) trees

```
*TypeDriven> semTrees s0
```

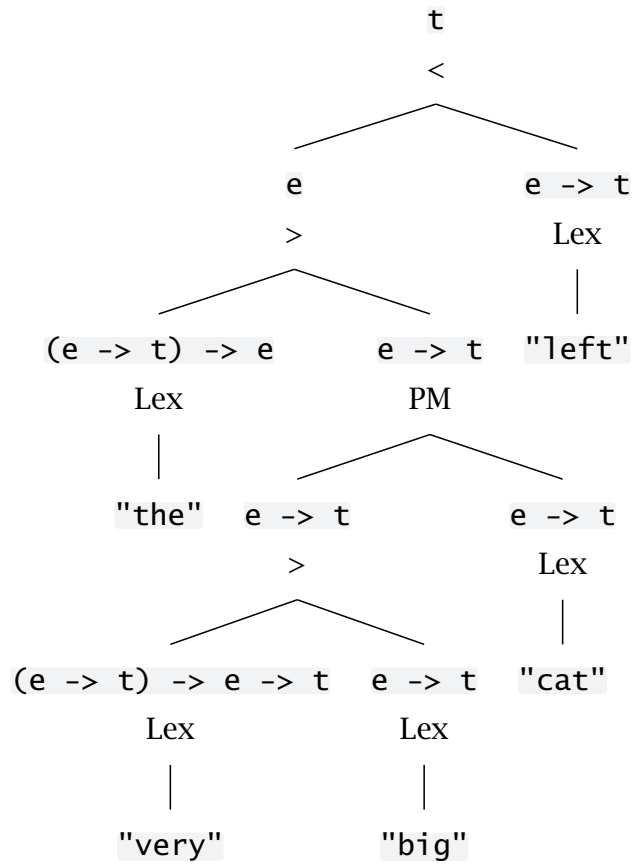


Semantic values as (syntax-homomorphic) trees

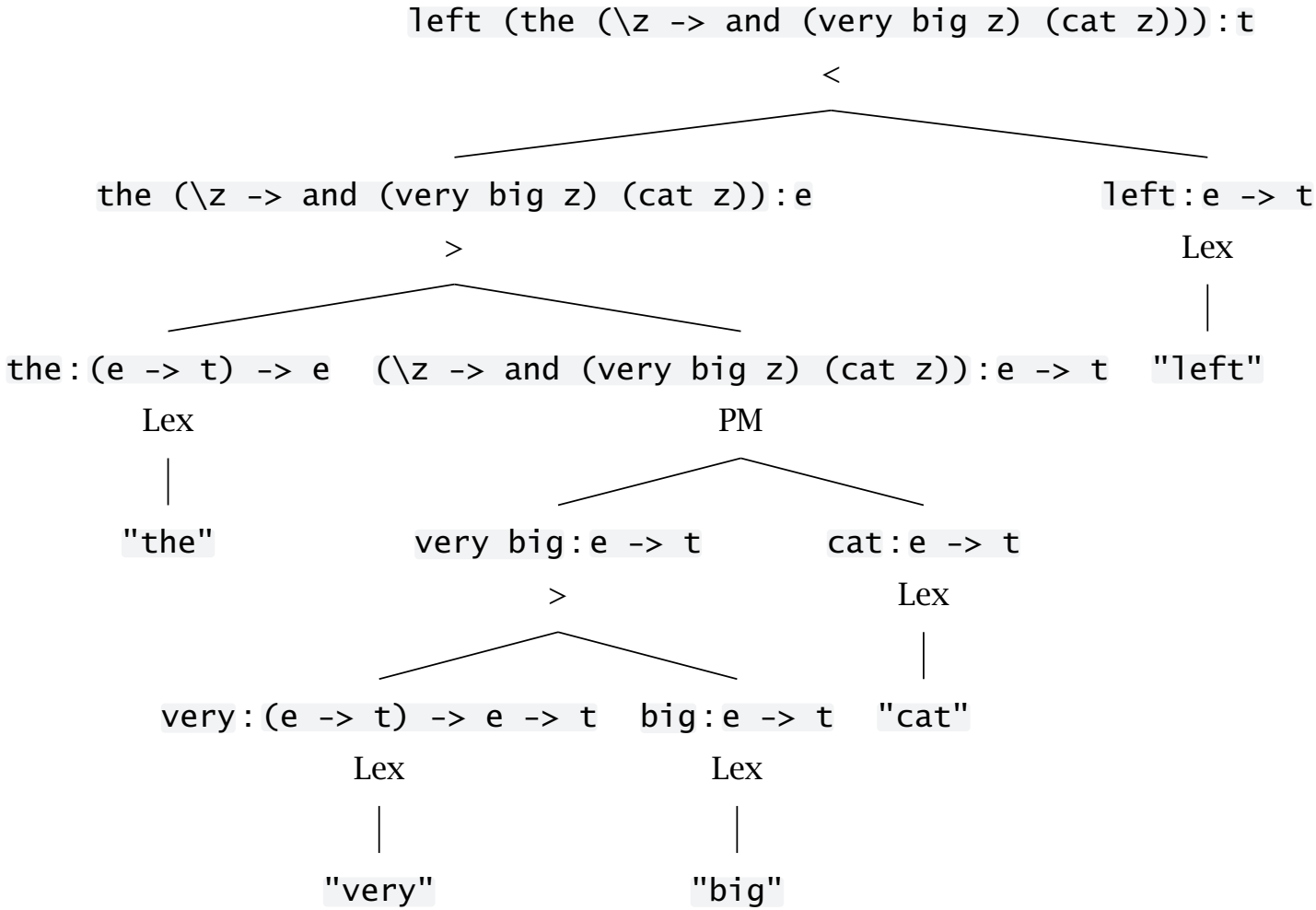
```
*TypeDriven> semTrees s0
```




```
*TypeDriven> semTrees s1
```



With some term normalization



Natural deduction

```
*TypeDriven> semProofs s0
```

$$\frac{\text{ann left} \vdash \texttt{left ann : t}}{\text{ann} \vdash \texttt{ann : e} \quad \text{left} \vdash \texttt{left : e -> t}}\ <$$

```
*TypeDriven> semProofs s1
```

$$\dfrac{\dfrac{\text{the} \vdash \texttt{the : (e -> t) -> e} \qquad \dfrac{\dfrac{\text{very} \vdash \texttt{very : (e -> t) -> e -> t} \qquad \text{big} \vdash \texttt{big : e -> t}}{\text{very big} \vdash \texttt{very big : e -> t}}\ > \qquad \text{cat} \vdash \texttt{cat : e -> t}}{\text{very big cat} \vdash \texttt{(\z -> and (very big z) (cat z)) : e -> t}}\ \text{PM}}{\text{the very big cat} \vdash \texttt{the (\z -> and (very big z) (cat z)) : e}} > \qquad \text{left} \vdash \texttt{left : e -> t}}{\text{the very big cat left} \vdash \texttt{left (the (\z -> and (very big z) (cat z))) : t}}\ <$$

Adding effectful things to the grammar

Regular types extended with effect-ful types:

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

Some notions of effects to get us going

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

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

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow \quad c \text{ , then } \begin{cases} F\,a \cdot \quad b \Rightarrow & F\,c \\ \quad a \cdot F\,b \Rightarrow & F\,c \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c), \text{ then } \begin{cases} F\,a \cdot \quad b \Rightarrow & F\,c \\ \quad a \cdot F\,b \Rightarrow & F\,c \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c), \text{ then } \begin{cases} F\,a \cdot \quad b \Rightarrow (\uparrow\mathbf{R}\, f\, l\, r := (\overbrace{\lambda\, l'.\, f\, l'\, r}^{a \to c}) \bullet l, F\, c) \\ a \cdot F\, b \Rightarrow (\uparrow\mathbf{L}\, f\, l\, r := (\underbrace{\lambda r'.\, f\, l\, r'}_{b \to c}) \bullet r, F\, c) \end{cases}$$

Extending `combine` with functors

Functorial F's don't disrupt whatever your semantics can already do:

$$\text{if } a \cdot b \Rightarrow (f, c), \text{ then } \begin{cases} F\,a \cdot \quad b \Rightarrow (\uparrow\mathbf{R}\, f\, l\, r := (\overbrace{\lambda\, l'.\, f\, l'\, r}^{a \to c}) \bullet l, F\, c) \\ a \cdot F\, b \Rightarrow (\uparrow\mathbf{L}\, f\, l\, r := (\underbrace{\lambda r'.\, f\, l\, r'}_{b \to c}) \bullet r, F\, c) \end{cases}$$

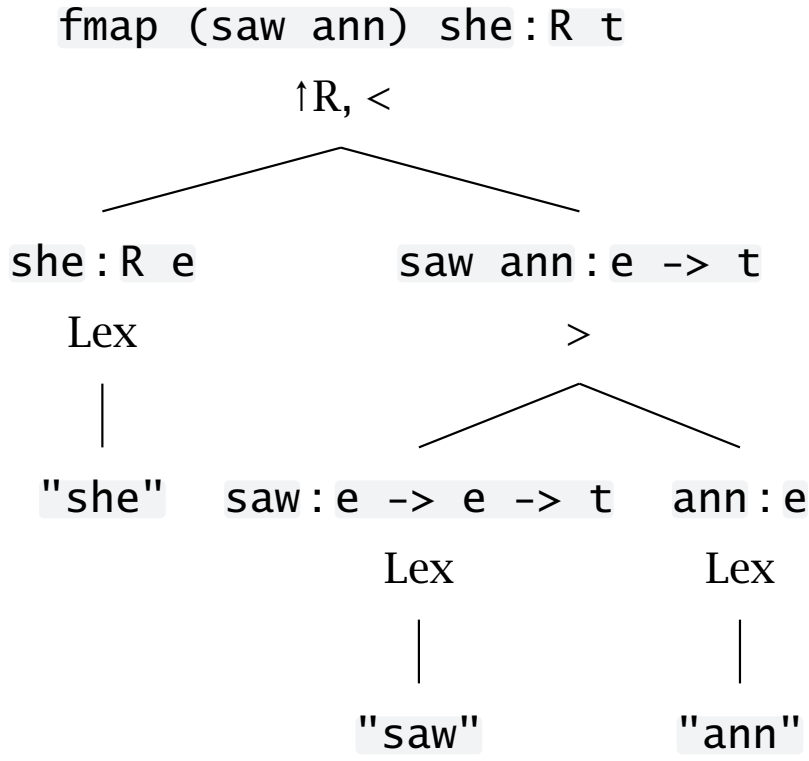
Ported directly to Haskell:

```
combine' :: Type -> Type -> [(Mode, Type)]
combine' l r = combine l r ++
  [ (LR op, Eff f c) | Eff f a <- [l]
                     , functor f, (op, c) <- combine' a r ] ++
  [ (LL op, Eff f c) | Eff f b <- [r]
                     , functor f, (op, c) <- combine' l b ]
```

This technique was developed by Barker & Shan 2014, White et al. 2017 for parsing with continuations. But continuations are functorial, and the technique works for any functor!

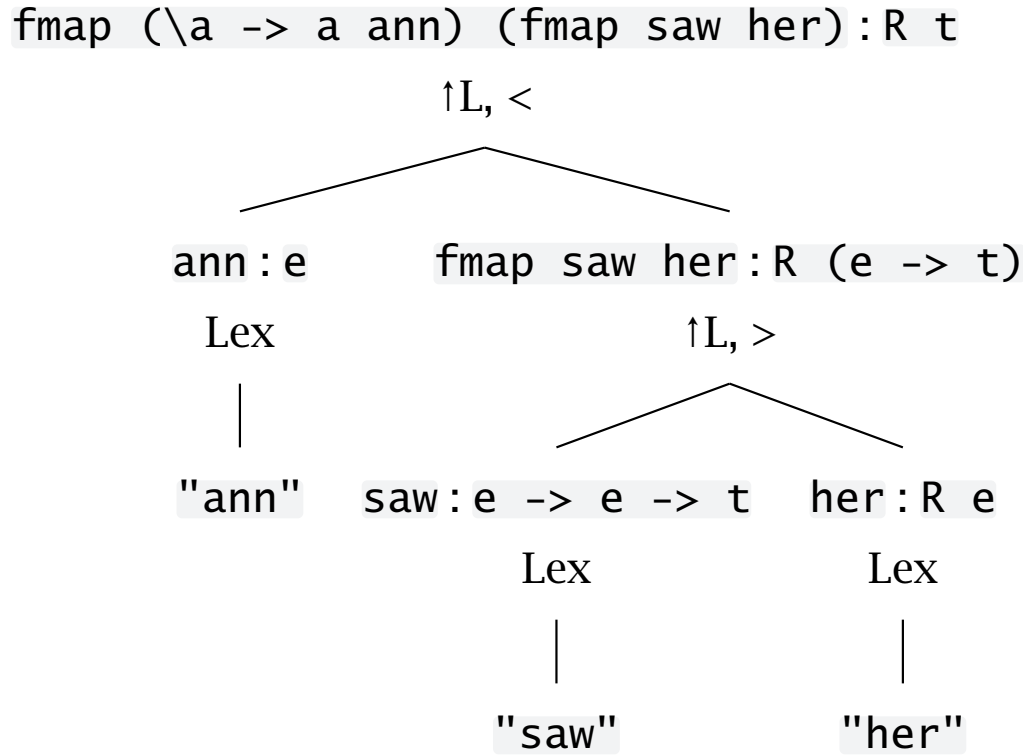
She saw Ann

```
*TDParse> semTrees (parse [she, saw, ann])
```



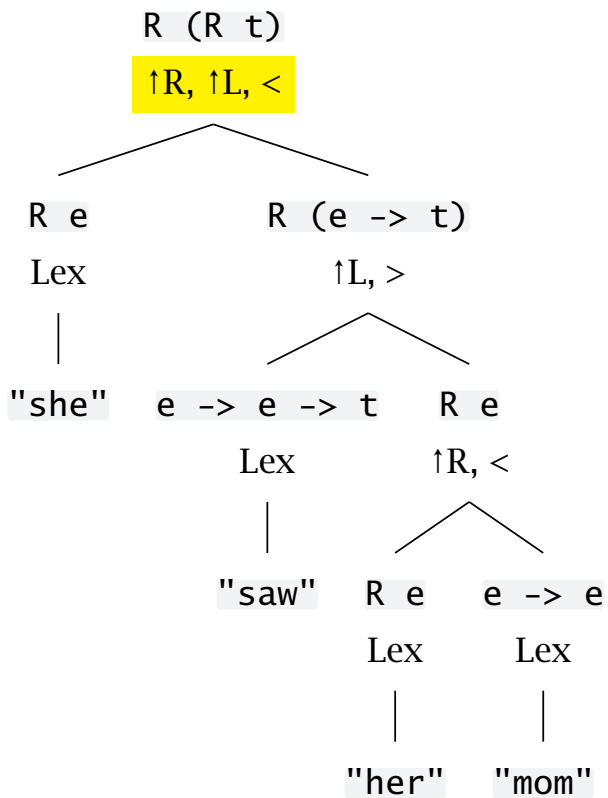
Ann saw her

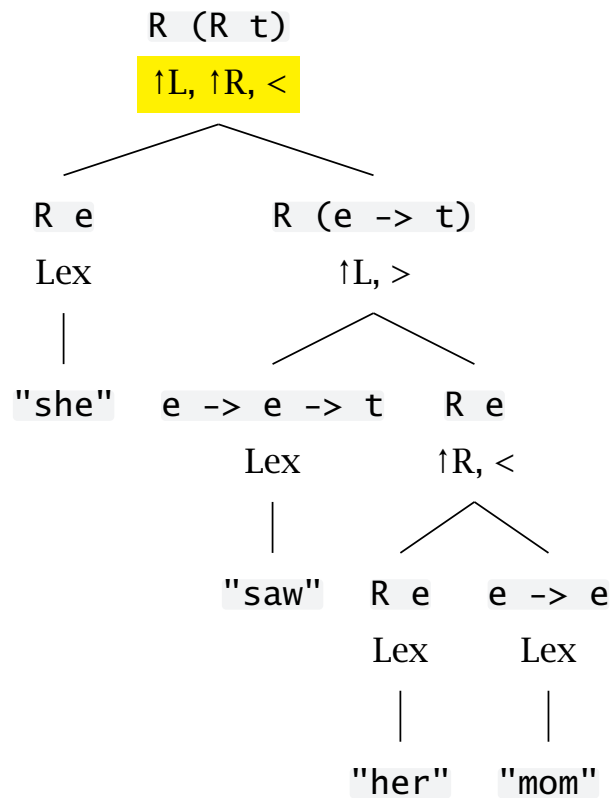
```
*TDParse> semTrees (parse [ann, saw, her])
```



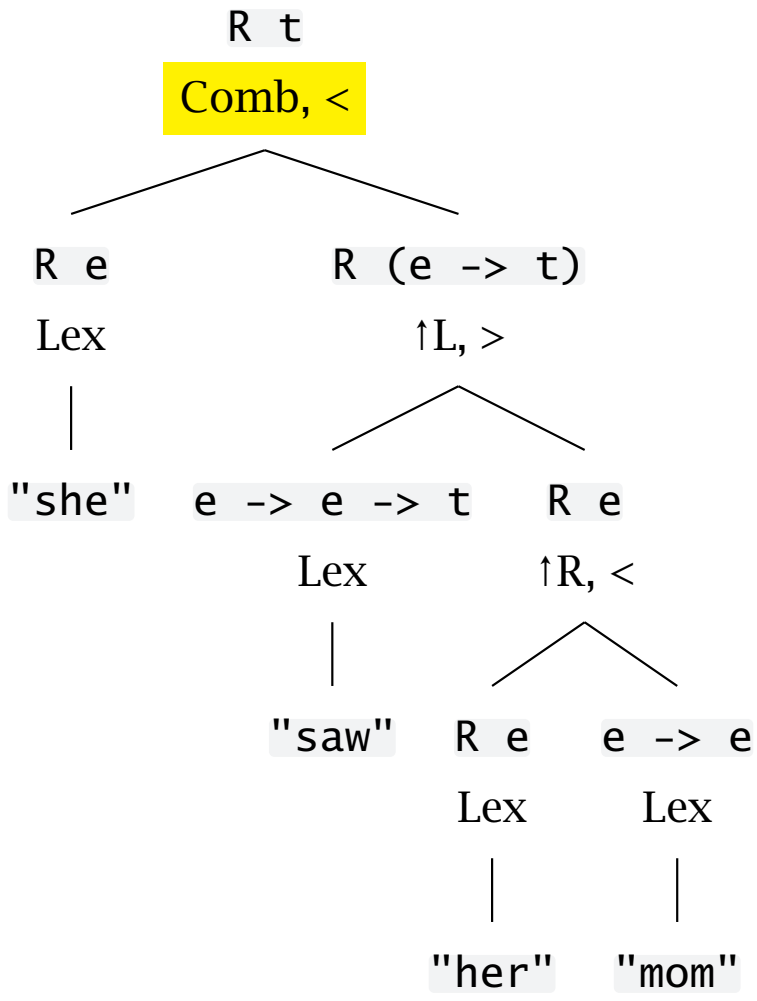
She saw her mom

```
*TDParse> semTrees (parse [she, saw, her, mom])
```





A mysterious third...



Some combinations

For any functors F, G, if $a \cdot b \Rightarrow c$, then:

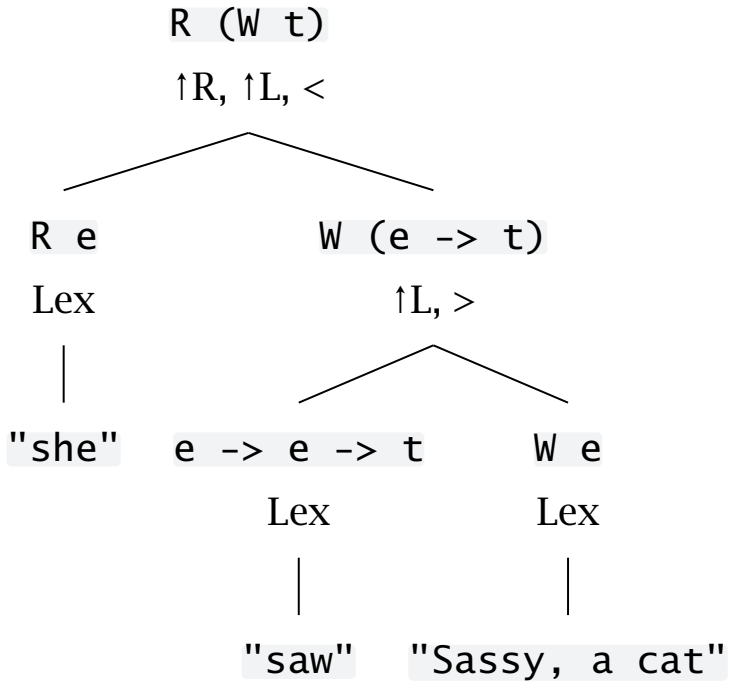
- $a \cdot G\,b \Rightarrow G\,c$
- $F\,a \cdot G\,b \Rightarrow F\,(G\,c)$

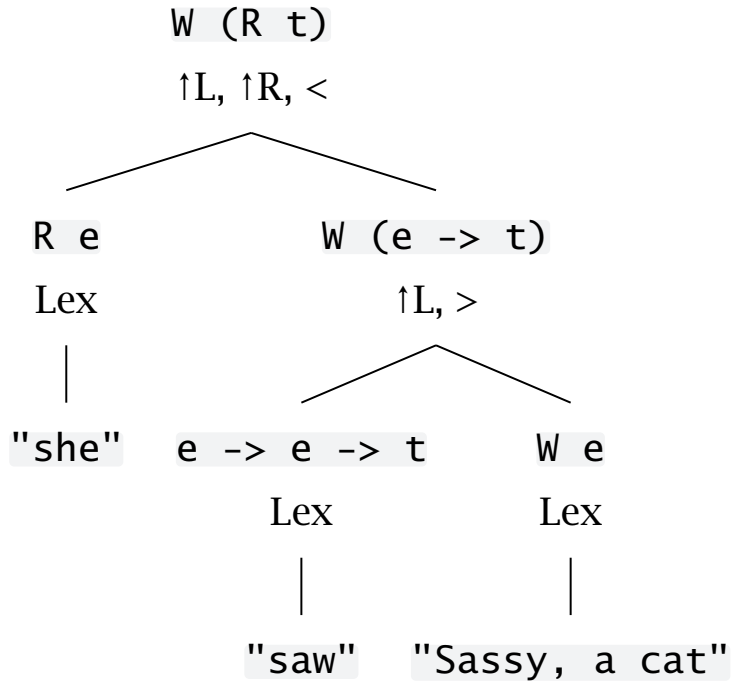
The reverse direction works as well:

- $F\,a \cdot b \Rightarrow F\,c$
- $F\,a \cdot G\,b \Rightarrow G\,(F\,c)$

F and G may be the same, or different.

Pronouns and other effects





Adding in Z

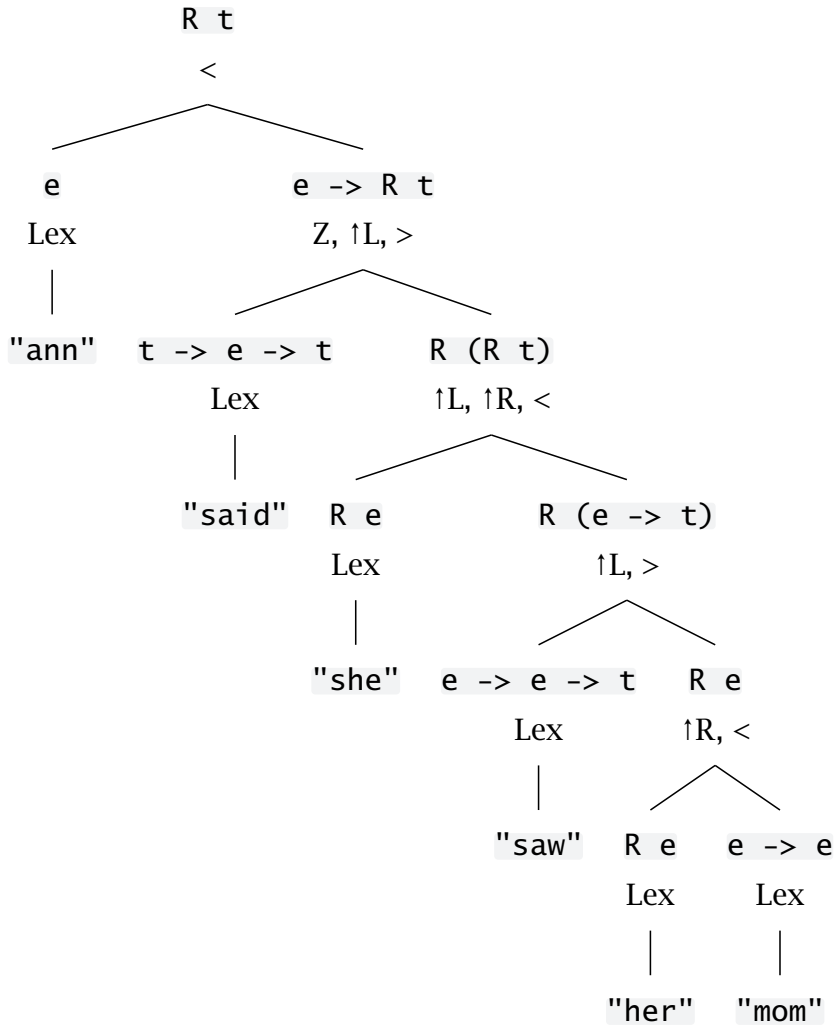
Reminding ourselves of Jacobson's Z rule for binding:

$$\frac{\lambda x.\, f\,(m\,x)\,x : \mathsf{e} \to b}{f : a \to \mathsf{e} \to b \qquad m : \mathsf{R}\,a}\ \mathsf{Z}$$

Straightforwardly imported to `combine` (and generalized a bit):

```
combine' l r =
-- ... ++
  [ (Z op, E :-> d) | a :-> E :-> b <- [l]
                    , Eff R c <- [r]
                    , (op, d) <- combine' (a :-> b) c ]
```

One Z-ful interpretation (of many)



This would be hair-raisingly complex in the original VFS architecture.

Barker, Chris & Chung-chieh Shan. 2014. *Continuations and natural language*. Oxford: Oxford University Press. https://doi.org/10.1093/acprof:oso/9780199575015.001.0001.

Charlow, Simon. 2019. A modular theory of pronouns and binding. Unpublished ms., Rutgers University. https://ling.auf.net/lingbuzz/003720.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar*. Oxford: Blackwell.

Jacobson, Pauline. 1999. Towards a variable-free semantics. *Linguistics and Philosophy* 22(2). 117–184. https://doi.org/10.1023/A:1005464228727.

White, Michael, Simon Charlow, Jordan Needle & Dylan Bumford. 2017. Parsing with dynamic continuized CCG. In *Proceedings of the 13th International Workshop on Tree Adjoining Grammars and Related Formalisms (TAG+13)*, 71–83. Umeå, Sweden: Association for Computational Linguistics. http://aclweb.org/anthology/W17-6208.