

# Effectful composition in natural language semantics

Applicatives and monads

Dylan Bumford (UCLA)   Simon Charlow (Yale)

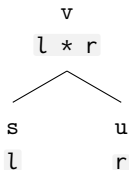
NASSLLI 2025 @ UW

June 26, 2025

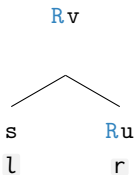
Functorial composition: Factoring out the patterns

## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$

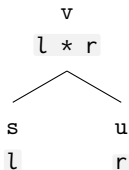


- ▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$

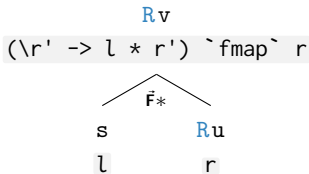


## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$

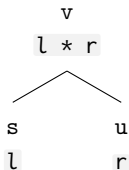


- ▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$



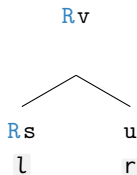
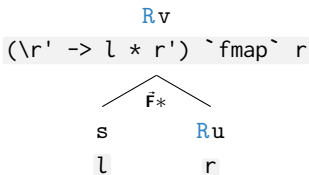
## Macro rules

Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



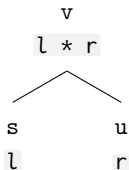
▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$

▷ What about when the left daughter is  $Rs$  and the right is  $u$

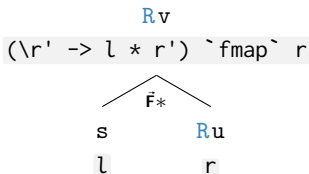


## Macro rules

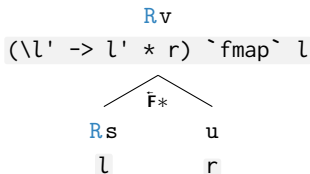
Let's say  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



- ▷ Can you combine when the left daughter is  $s$  and the right is  $Ru$



- ▷ What about when the left daughter is  $Rs$  and the right is  $u$

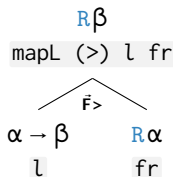
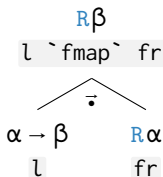


## Modes from modes

This schema gives us a way to make new modes of combination out of old ones!

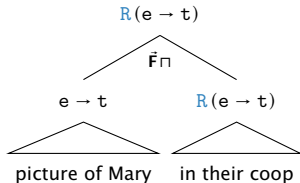
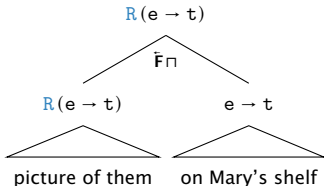
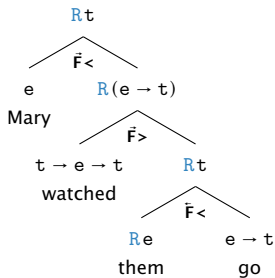
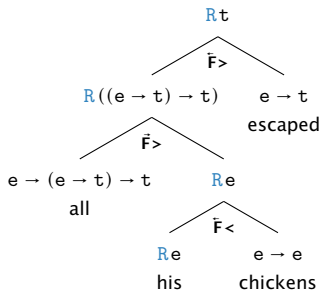
```
mapL (*) fl r = (\l -> l * r) `fmap` fl
mapR (*) l fr = (\r -> l * r) `fmap` fr
```

For instance, `mapL (>)` is equivalent to the mapping mode  $\vec{\cdot}$



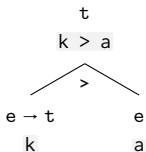
```
mapL (>) l fr = (\r -> l > r) `fmap` fr
--           = (\r -> l r) `fmap` fr
--           = l `fmap` fr
```

# In action

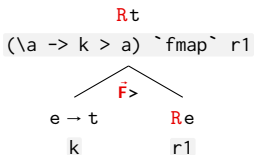


# Recurring on the right

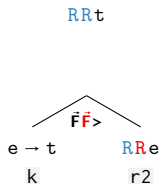
## 1st-order MoC



## 2nd-order MoC

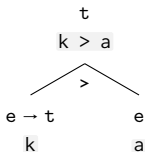


## 3rd-order MoC

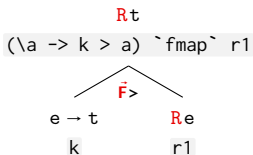


# Recurring on the right

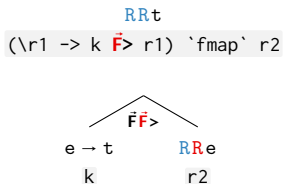
## 1st-order MoC



## 2nd-order MoC

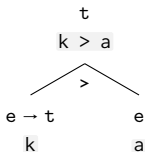


## 3rd-order MoC

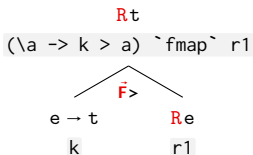


# Recurring on the right

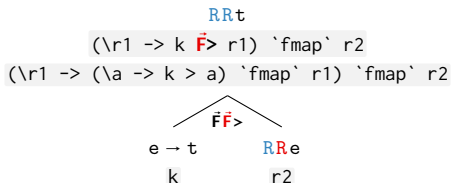
## 1st-order MoC



## 2nd-order MoC

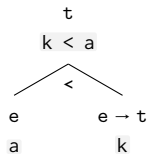


## 3rd-order MoC

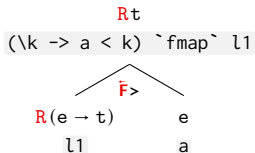


# Recurring on the left

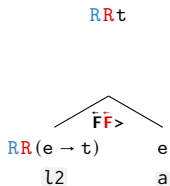
## 1st-order MoC



## 2nd-order MoC

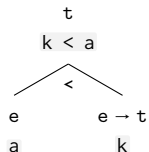


## 3rd-order MoC

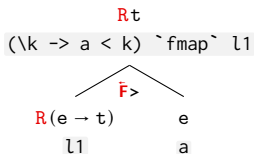


# Recurring on the left

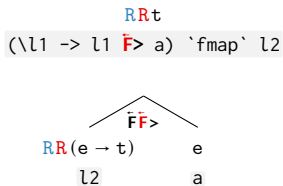
## 1st-order MoC



## 2nd-order MoC

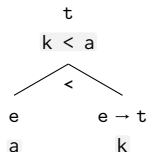


## 3rd-order MoC

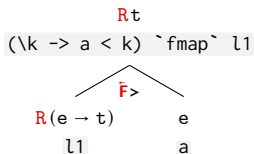


# Recurring on the left

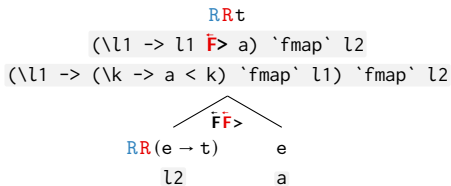
## 1st-order MoC



## 2nd-order MoC



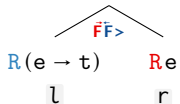
## 3rd-order MoC



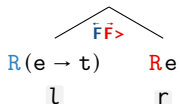
## Recurring on both sides

What happens when both daughters are functorial?

R R t



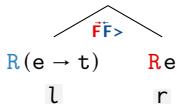
R R t



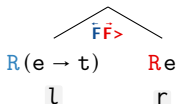
## Recurring on both sides

What happens when both daughters are functorial?

$\text{RRt}$   
`(\a -> l  $\tilde{\text{F}}$ > a) `fmap` r`



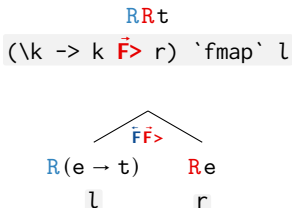
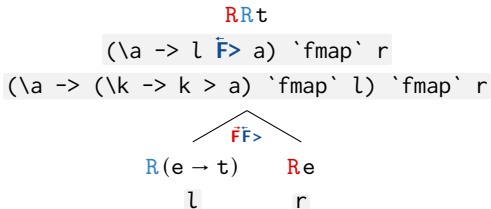
$\text{RRt}$





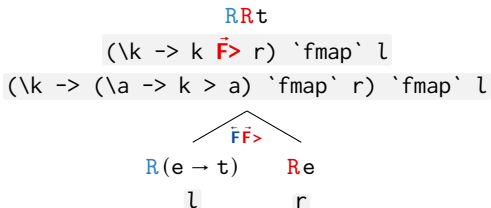
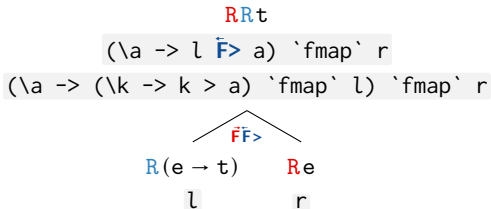
## Recurring on both sides

What happens when both daughters are functorial?



## Recurring on both sides

What happens when both daughters are functorial?



## Comments, questions, concerns addressed

Are occurrences of • actually instantiated in the syntax?

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

How is this *practical*? < 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.  
Derivations can be completely mechanized

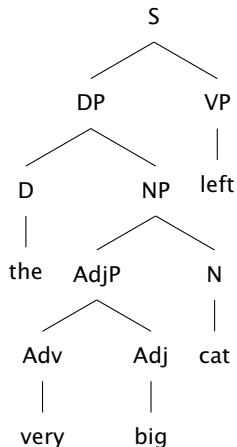
[schar.github.io/TDParse](https://schar.github.io/TDParse)

## Mechanizing composition

## 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  
| Comb Mode Sem Sem

**data** Mode

= FA | BA  
| PM -- etc

**data** Type

= E | T  
| Type :-> Type

## Implementing type-driven combination

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

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

```
basic 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

```
combine = basic
```

```
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) =
  [(Comb 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
```

```
(Comb BA (Lex "ann") (Lex "left"), T)
```

```
ghci> synsem s1
```

```
((Comb BA (Comb FA (Lex "the") (Comb PM (Comb FA (Lex "very")
(Lex "big"))) (Lex "cat")))) (Lex "left")), T)
```

## Adding effectful things to the grammar

Some notions of computational effects to get us going

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

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

Regular types extended with computational types:

```
data Type = E | T  
          | Type -> Type  
          | Comp Eff 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 = basic l r
```

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

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

## Some combine-ations

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

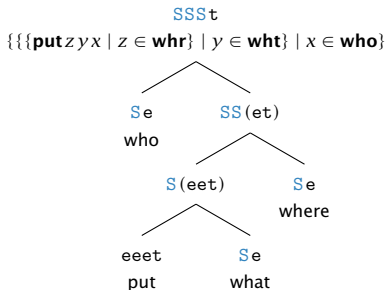
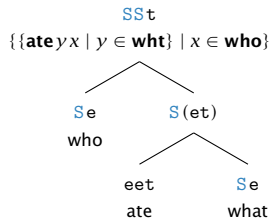
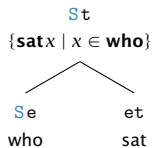
```
ghci> combine E (Comp R (E :-> T))
[(MR BA, Comp R T)]
```

```
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))]
```

# Applicatives

# Effectssssss



We've been celebrating the success of the recursive mapping mode for its ability to handle all of these situations without breaking a sweat...

... but is there anything you might be worried about?

## Unselective association

It's perhaps disconcerting that all of these questions should have *necessarily* different types

Indeed, closure operators that associate with questions **don't distinguish between them**; for instance:

- (1) Mary knows who sat
- (2) Mary knows who ate what
- (3) Mary knows who put what where

Assuming 'knows' ::  $S t \rightarrow e \rightarrow t$ , we have no trouble putting these sentences together, but they don't mean what you expect. . .

## Unselectivity writ large

The problem is quite general

'only' ::  $F(et) \rightarrow (et)$

- (4) Mary only introduced JENNIFER to Bill
- (5) Mary only introduced Jennifer to BILL
- (6) Mary only introduced JENNIFER to BILL

'can' ::  $S t \rightarrow t$

- (7) You can eat any apple
- (8) You can eat any apple anywhere you like

'as for Mary' ::  $R t \rightarrow t$

- (9) As for Mary, she called yesterday
- (10) As for Mary, she called her dad
- (11) As for Mary, she called her dad on her birthday

## Flatter meanings

It seems that in general we want these sorts of operators to be able to handle multiple effects simultaneously

But the mapping modes we have so far necessarily separate each effect into its own stratum

In fact, if you've done any textbook natural language semantics for pronouns or questions, you already know what sorts of meanings we want here:

- ▶  $\llbracket \text{who ate what} \rrbracket = \{\mathbf{ate} y x \mid y \in \mathbf{wht}, x \in \mathbf{who}\}$
- ▶  $\llbracket \text{she}_0 \text{ met him}_4 \rrbracket = \lambda g. \mathbf{met} g_4 g_0$

# Hamblin Semantics

## Types:

$\zeta ::= e \mid t \mid \dots$  (Base pre-types)

$\mid \zeta \rightarrow \zeta$  (Function pre-types)

$\tau ::= S\zeta$  (Expression types)

## Lexicon:

who ::  $S e$   
:=  $\{x \mid \mathbf{person} x\}$

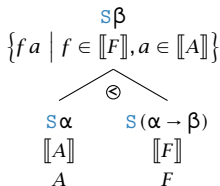
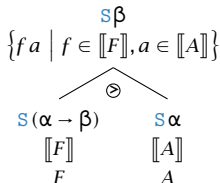
Mars ::  $S e$   
:=  $\{m\}$

cat ::  $S(e \rightarrow t)$   
:=  $\{\mathbf{cat}\}$

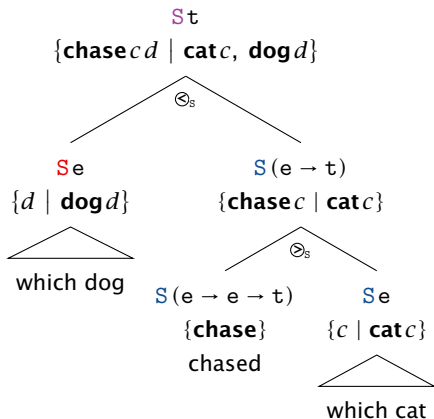
nobody ::  $S((e \rightarrow t) \rightarrow t)$   
:=  $\{\mathbf{nobody}\}$

...

## Pointwise Function Application



## A Hamblin derivation



# Heim & Kratzer

## Types:

$\zeta ::= e \mid t \mid \dots$  (Base pre-types)

$\mid \zeta \rightarrow \zeta$  (Function pre-types)

$\tau ::= \mathbb{R}\zeta$  (Expression types)

## Lexicon:

$it_n :: \mathbb{R}e$   
 $:= \lambda i. i_n$

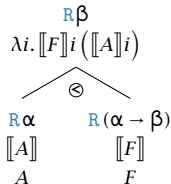
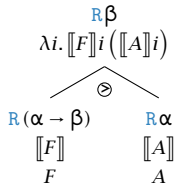
$Mars :: \mathbb{R}e$   
 $:= \lambda i. m$

$cat :: \mathbb{R}(e \rightarrow t)$   
 $:= \lambda i. cat$

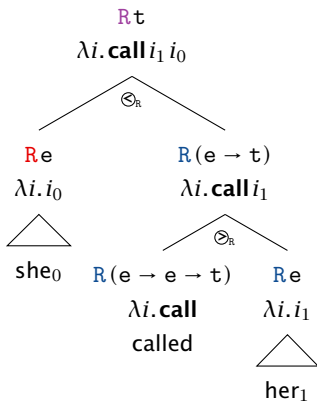
$nobody :: \mathbb{R}((e \rightarrow t) \rightarrow t)$   
 $:= \lambda i. nobody$

...

## Env.-Sensitive Function Application

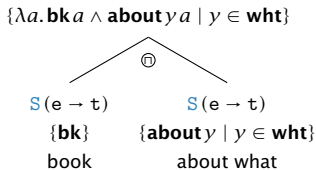
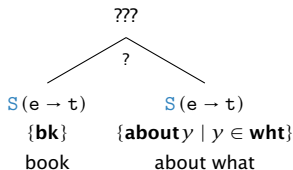


## A Heim & Kratzer derivation

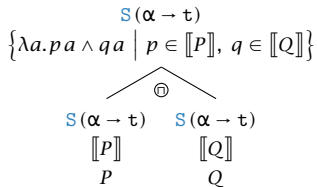


## Pointwise everything

Just as before though, these special-purpose applicative modes do not generalize: every mode of combination needs to be redefined

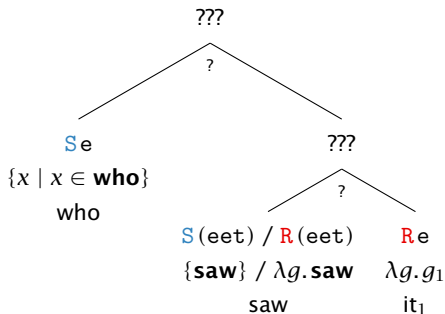


### Pointwise Modification



## Effect cross-pollination

Much more troublingly, there is no way to combine daughters that generate *different effects*! The two grammars are entirely quarantined



Clearly we want something a little more flexible and a little more general

## Factoring out the pattern

The Hamblin grammar and the Heim & Kratzer grammar make use of the following components:

- ▶ Uniformity: ordinary values are **injected** into the enriched type
- ▶ Merger: application **amalgamates** the effects of the daughters

### Hamblin decomposed

$$\eta x := \{x\}$$

$$F \circledast A := \{f a \mid f \in F, a \in A\}$$

### Heim & Kratzer decomposed

$$\eta x := \lambda r. x$$

$$F \circledast A := \lambda r. Fr(Ar)$$

# Applicatives

This design pattern is so common that it too has a canonical name: an **Applicative Functor**

Formally, a constructor  $F$  is applicative if there are  $\eta$  and  $\circledast$  operations such that:

$$\eta :: a \rightarrow Fa$$

$$\circledast :: F(a \rightarrow b) \rightarrow Fa \rightarrow Fb$$

## Homomorphism

$$\eta f \circledast \eta x = \eta (fx)$$

## Identity

$$\eta (\lambda x. x) \circledast v = v$$

## Interchange

$$\eta (\lambda f. fx) \circledast u = u \circledast \eta x$$

## Composition

$$\eta (\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$$

These laws guarantee that  $\eta$  is a trivial way to inject something into the richer  $F$  type, and  $\circledast$  is function application lifted into  $F$

# Applicatives in Haskell

Haskell, too, provides a canonical type class for effects in which this sort of amalgamation is possible:

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

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The compiler will ensure that the operations you provide are appropriately typed, but it's your job to make sure they're well-behaved.

As the name suggests, if  $F$  is an applicative functor, then it is a functor...

`fmap`  $f$   $m =$

## Applicatives in Haskell

Haskell, too, provides a canonical type class for effects in which this sort of amalgamation is possible:

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

class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

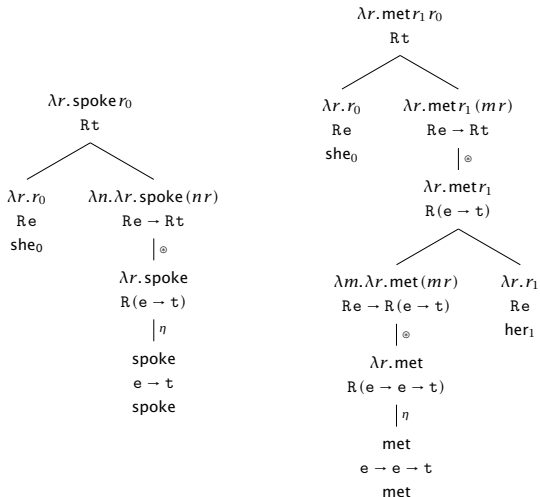
The compiler will ensure that the operations you provide are appropriately typed, but it's your job to make sure they're well-behaved.

As the name suggests, if  $F$  is an applicative functor, then it is a functor...

$$\text{fmap } f \ m = \text{pure } f \ \langle * \rangle \ m$$

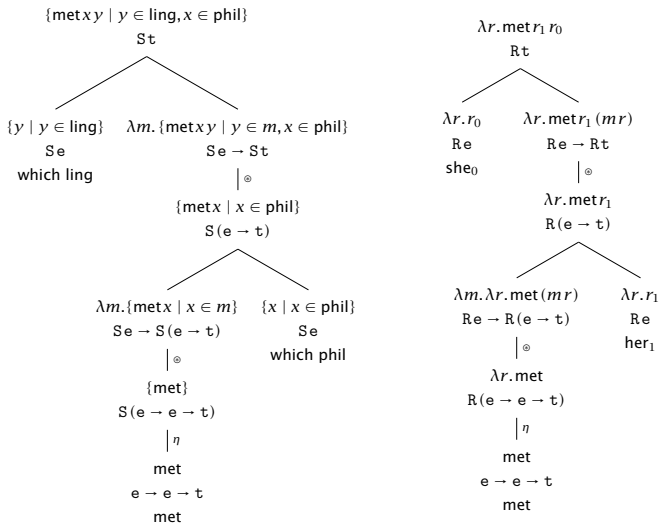
## Running with $\eta$ and $\circledast$

As with the discovery that some meanings were functorial, we could, if we like, immediately put these raw operators straight to work:



# Symmetry

And of course, any derivation for one effect will serve as a template for any other



## Intonational focus

In fact, all of the effects we introduced on Day 1 are applicative!

For instance, here are the operators for the focus type:  $Fa ::= a \times Sa$

$$\eta x := \langle x, \{x\} \rangle \quad (f, S) \otimes (x, T) := (fx, \{st \mid s \in S, t \in T\})$$

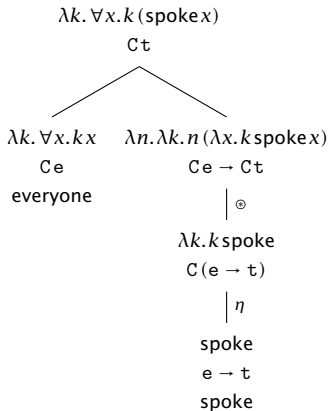
Using this applicative, we can derive the following meanings:

12. Mary introduced JENNIFER to Bill.  $\rightsquigarrow \{\text{intro } x b m \mid x \in \text{alt}_j\}$
13. Mary introduced Jennifer to BILL.  $\rightsquigarrow \{\text{intro } j y m \mid y \in \text{alt}_b\}$
14. Mary introduced JENNIFER to BILL.  $\rightsquigarrow \{\text{intro } x y m \mid x \in \text{alt}_j, y \in \text{alt}_b\}$

## Scope and continuations

Even the continuation functor for modeling expressions that take control of their syntactic contexts:  $C a ::= (a \rightarrow t) \rightarrow t$

$$\eta x := \lambda k. k x \quad m \circledast n := \lambda k. m (\lambda f. n (\lambda x. k (f x)))$$



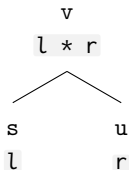
## Applicative-driven composition

But... we're back to the cat



## Applicativizing an underlying mode

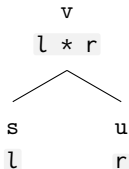
Imagine  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



- ▶ Can you combine when the left daughter is  $\Sigma s$  and the right is  $\Sigma u$ ?

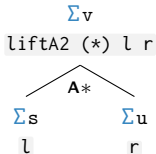
## Applicativizing an underlying mode

Imagine  $(*) :: s \rightarrow u \rightarrow v$  is a mode of combination that puts together a left daughter of type  $s$ , a right daughter of type  $u$



▷ Can you combine when the left daughter is  $\Sigma s$  and the right is  $\Sigma u$ ?

### liftA2



```
liftA2 :: Applicative f => ( s -> u -> v )
                               -> ( f s -> f u -> f v )
liftA2 (*) l r = pure (*) <*> l <*> r
--                = fmap (*) l <*> r
```

## Extending combine with applicatives

Ported directly to Haskell, again:

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

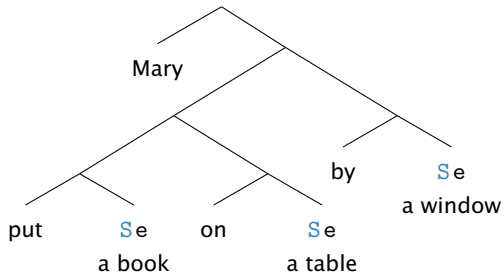
```
combine :: Type -> Type -> [(Mode, Type)]
combine l r = basic l r ++
  [ (MR op, Comp f c) | Comp f a <- [l]
                               , functor f, (op, c) <- combine a r ] ++
  [ (ML op, Comp f c) | Comp f b <- [r]
                               , functor f, (op, c) <- combine l b ] ++
  [ (AP op, Comp f c) | Comp f a <- [l], Comp g b <- [r], f == g
                               , applicative f, (op, c) <- combine a b ]
```

... And that's it! The derivations will now find themselves

# Monads

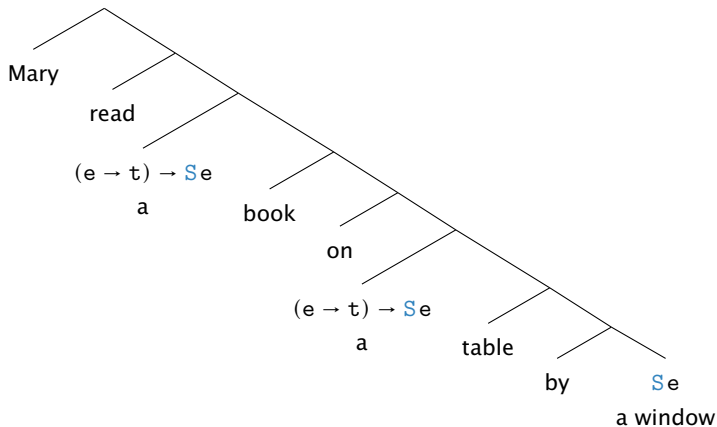
## Independent effects

(15) Mary put the book on the table by the window



## Dependent effects

(16) Mary read the book on the table by the window



## Syntactic nesting

There is, oddly, a predicted difference between the possible interpretations of *syntactically independent* effects and *syntactically nested* effects

- ▶ the former can result in a flat first-order computation
- ▶ the latter is obligatorily higher-order

And naturally this prediction will be generalized across all effects

- (17) a. Mary put the book on the table Mt  
b. Mary read the book on the table MMt
- (18) a. At least one student attended every class Ct  
b. At least one student in every class stayed awake CCt
- (19) a. Mary or John coughed or laughed at me St  
b. Mary coughed or laughed at me or John SSt

Does this seem right?

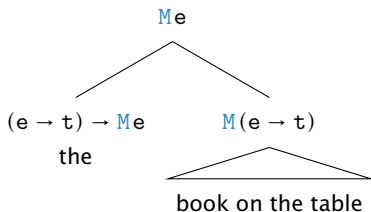
## Handling nested effects

Doesn't seem like it!

- (20) Bill knows which member of which committee I should talk to
- (21) We have neither the organizer nor a student from every class
- (22) If Mary coughed or laughed at me or John, run

In all of these cases, the relevant operator — 'knows', 'nor', 'if' — can handle all of the effects, even though they're nested

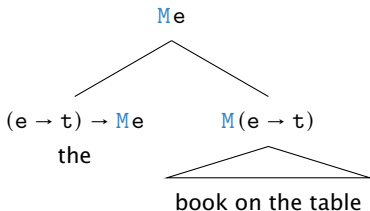
## Solving the types



If we want to be able to interpret this expression as presupposing a single composite domain condition, then we'll need a way to combine:

$$\text{'the'} :: (e \rightarrow t) \rightarrow Me \quad + \quad \text{'book on the table'} :: M(e \rightarrow t) \quad = \quad Me$$

## Solving the types



If we want to be able to interpret this expression as presupposing a single composite domain condition, then we'll need a way to combine:

'the' ::  $(e \rightarrow t) \rightarrow Me$     +    'book on the table' ::  $M(e \rightarrow t)$     =     $Me$

```
(??) :: ( (E -> T) -> Maybe E ) -> Maybe (E -> T) -> Maybe E
the ?? bkonthetab = case bkonthetab of
  Nothing -> Nothing
  Just p   -> the p
```

## Solving more types

How about the other effects

$$\text{'some'} :: (e \rightarrow t) \rightarrow \mathbf{S}e \quad + \quad \text{'book on some table'} :: \mathbf{S}(e \rightarrow t) \quad = \quad \mathbf{S}e$$

## Solving more types

How about the other effects

'some' :: (e → t) → S e + 'book on some table' :: S (e → t) = S e

```
(??) :: ( (E -> T) -> [E] ) -> [E -> T] -> [E]  
some ?? bkonsometab = concat [some p | p <- bkonsometab]
```

## Solving more types

How about the other effects

'some' ::  $(e \rightarrow t) \rightarrow S e$  + 'book on some table' ::  $S(e \rightarrow t)$  =  $S e$

```
(??) :: ( (E -> T) -> [E] ) -> [E -> T] -> [E]  
some ?? bkonsometab = concat [some p | p <- bkonsometab]
```

'every' ::  $(e \rightarrow t) \rightarrow C e$  + 'book on every table' ::  $C(e \rightarrow t)$  =  $C e$

## Solving more types

How about the other effects

'some' ::  $(e \rightarrow t) \rightarrow S e$  + 'book on some table' ::  $S(e \rightarrow t)$  =  $S e$

```
(??) :: ( (E -> T) -> [E] ) -> [E -> T] -> [E]
some ?? bkonsometab = concat [some p | p <- bkonsometab]
```

'every' ::  $(e \rightarrow t) \rightarrow C e$  + 'book on every table' ::  $C(e \rightarrow t)$  =  $C e$

```
(??) :: ( (E -> T) -> Cont E ) -> Cont (E -> T) -> Cont E
every ?? (Cont bkoneverytab) =
  Cont (\k -> bkoneverytab (\p -> let Cont q = every p in q k))
```

## Generalizing the solution

Actually, all of these definitions have more general applicability, not just in the case where the left thing expects a property and the right thing computes a property

The same functions will work fine for any type  $a$  in place of  $E \rightarrow T$ , and actually any type  $b$  in place of  $E$

```
(??) :: (a -> Maybe b) -> Maybe a -> Maybe b
f ?? m = case m of
  Nothing -> Nothing
  Just p   -> f p
```

```
(??) :: (a -> [b]) -> [a] -> [b]
f ?? m = concat [f p | p <- m]
```

```
(??) :: (a -> Cont b) -> Cont a -> Cont b
f ?? (Cont m) = Cont (\k -> m (\p -> let Cont q = f p in q k))
```

## You could have invented Monads

As you might begin to expect, this pattern is so general, it has a name

A type constructor  $F$  is called a **Monad** if there are  $\eta$  and  $\llcorner$  such that:

$$\eta :: a \rightarrow Fa \qquad \llcorner :: (a \rightarrow Fb) \rightarrow Fa \rightarrow Fb$$

**Left Identity**  $k \llcorner \eta a = k a$

**Right Identity**  $\eta \llcorner m = m$

**Associativity**  $k \llcorner (c \llcorner m) = (\lambda x. k \llcorner c x) \llcorner m$

The  $\eta$  plays exactly the same role as it did for the Applicative computations

And indeed, all of the effects we've seen are also Monads

## Monads in Haskell

Haskell of course also comes with a pre-defined type class for effects of this sort:

```
class Applicative f => Monad f where
  return :: a -> f a
  (>>=) :: f a -> (a -> f b) -> f b
```

- ▶ For purely historical reasons, `pure` gets a synonym called `return`
- ▶ For purely ergonomic reasons, it flips the args of `≡`:

E.g.,

```
instance Monad [] where
  return x = [x]
  m >>= k = concat [k x | x <- m]
```

## Join

As usual, we could throw this new polymorphic operator directly into the system and get to work, but in this case there's a shortcut

From  $\bowtie$ , we can define another operation:

$$\begin{aligned}\mu &:: \Sigma \Sigma \alpha \rightarrow \Sigma \alpha \\ \mu M &:= \mathbf{id} \bowtie M\end{aligned}$$

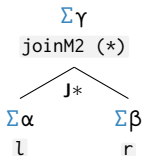
For any monadic computation  $\Sigma$ , this simply flattens two layers of effects into one composite effect

```
join :: Monad f => f (f a) -> f a
join m = m >>= id
```

And remember that we're already getting an awful lot of these nested structures from all the maps

## Join as a mode of combination

So all we really need is a way to squash any doubly-layered effects that we happen to build up in the course of composition



```
joinM2 :: Monad f => (a -> b -> f (f c)) ->
                    (a -> b -> f c)
joinM2 (*) l r = join (l * r)
```

## Type-driven monads

Can this operation be type-driven? Certainly:

```
combine :: Type -> Type -> [(Mode, Type)]
combine l r = reduce $
  basic l r ++
  [ (MR op, Comp f c) | Comp f a <- [l]
    , functor f, (op, c) <- combine a r ] ++
  [ (ML op, Comp f c) | Comp f b <- [r]
    , functor f, (op, c) <- combine l b ] ++
  [ (AP op, Comp f c) | Comp f a <- [l], Comp g b <- [r], f == g
    , applicative f, (op, c) <- combine a b ]

reduce combos =
  combos ++
  [ (JN op, Comp f a) | (op, Eff f (Eff g a)) <- combos
    , f == g, monad f ]
```

## Monads and Applicatives

Notice we now have three ways to compose when both sides are semantically interesting:

$$\begin{array}{c} \Sigma\gamma \\ \swarrow \quad \searrow \\ \text{JFF}^* \\ \swarrow \quad \searrow \\ \Sigma\alpha \quad \Sigma\beta \\ \text{L} \quad \text{R} \end{array} = \text{join} \left( \begin{array}{c} \Sigma\Sigma\gamma \\ \swarrow \quad \searrow \\ \text{FF}^* \\ \swarrow \quad \searrow \\ \Sigma\alpha \quad \Sigma\beta \\ \text{L} \quad \text{R} \end{array} \right)$$

$$\begin{array}{c} \Sigma\gamma \\ \swarrow \quad \searrow \\ \text{A}^* \\ \swarrow \quad \searrow \\ \Sigma\alpha \quad \Sigma\beta \\ \text{L} \quad \text{R} \end{array}$$

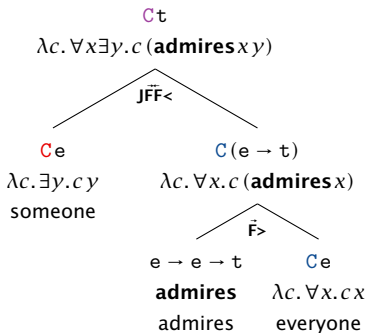
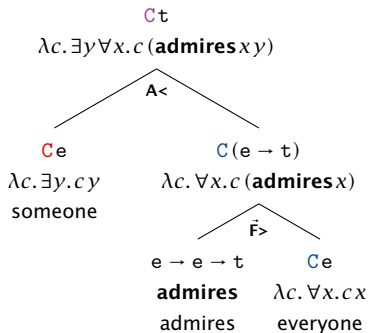
$$\begin{array}{c} \Sigma\gamma \\ \swarrow \quad \searrow \\ \text{JFF}^* \\ \swarrow \quad \searrow \\ \Sigma\alpha \quad \Sigma\beta \\ \text{L} \quad \text{R} \end{array} = \text{join} \left( \begin{array}{c} \Sigma\Sigma\gamma \\ \swarrow \quad \searrow \\ \text{FF}^* \\ \swarrow \quad \searrow \\ \Sigma\alpha \quad \Sigma\beta \\ \text{L} \quad \text{R} \end{array} \right)$$

By convention, the join instance for a given effect should make the first two options equivalent

But the third derivation provides a meaning we haven't yet been able to capture **inverse scope**

## Inverse scope

Thus we have the following two derivations:



Notice that the second one requires a more complicated mode of combination at the top node, as perhaps you might hope

