Fragment Grammars: Productivity and Reuse in Language

Timothy J. O'Donnell
MIT

-ness

-ness

- *circuitousness*, *grandness*, *orderliness*, *pretentiousness*, *cheapness*, *coolness*, *warmness*, ...

-ness

- *circuitousness*, *grandness*, *orderliness*, *pretentiousness*, *cheapness*, *coolness*, *warmness*, ...
- Adj>N

-ness

- *circuitousness*, *grandness*, *orderliness*, *pretentiousness*, *cheapness*, *coolness*, *warmness*, ...
- Adj>N
- *grand* + -ness

-ness

- *circuitousness*, *grandness*, *orderliness*, *pretentiousness*, *cheapness*, *coolness*, *warmness*, ...
- Adj>N
- *grand* + -ness
- *pine-scentedness*

-ity

-ity

- *verticality*, *tractability*, *severity*, *seniority*, *inanity*, *electricity*, ...

-ity

- *verticality*, *tractability*, *severity*, *seniority*, *inanity*, *electricity*, ...
- Adj>N

-ity

- *verticality*, *tractability*, *severity*, *seniority*, *inanity*, *electricity*, ...
- Adj>N
- Stress change (e.g., *normalness* v. *normality*), vowel laxing (e.g., *inane* v. *inanity*)

-ity

- *verticality*, *tractability*, *severity*, *seniority*, *inanity*, *electricity*, ...
- Adj>N
- Stress change (e.g., *normalness* v. *normality*), vowel laxing (e.g., *inane* v. *inanity*)
- *The red lantern indicated the ethnicity/ ethnicness of the restaurant*

-ity

- *verticality*, *tractability*, *severity*, *seniority*, *inanity*, *electricity*, ...
- Adj>N
- Stress change (e.g., *normalness* v. *normality*), vowel laxing (e.g., *inane* v. *inanity*)
- *The red lantern indicated the ethnicity/ ethnicness of the restaurant*
- **pine-scentedity*

-ity

-ity

- But ...

-ity

- But ...
  - -ile/-al/-able/-ic/-(i)an

-ity

- But ...
  - -ile/-al/-able/-ic/-(i)an
  - *Bayesable*

-ity

- But ...
  - -ile/-al/-able/-ic/-(i)an
  - *Bayesable*
    - *Bayesability*

-ity

- But ...
  - -ile/-al/-able/-ic/-(i)an
  - *Bayesable*
    - *Bayesability*
  - *Coolity is not trying* (from *Huffington Post*)

-th

-th

- *warmth*, *width*, *truth*, *depth*, ...

-th

- *warmth*, *width*, *truth*, *depth*, ...
- Adj>N

-th

- *warmth*, *width*, *truth*, *depth*, ...
- Adj>N
- *heal/health*, *dead/death*, *young/youth*, *vile/filth*, *slow/sloth*

-th

- *warmth*, *width*, *truth*, *depth*, ...
- Adj>N
- *heal/health*, *dead/death*, *young/youth*, *vile/filth*, *slow/sloth*
- *weal?/wealth*, *?wroth/wrath*, *?merry/mirth*

-th

- *warmth*, *width*, *truth*, *depth*, ...
- Adj>N
- *heal/health*, *dead/death*, *young/youth*, *vile/filth*, *slow/sloth*
- *weal?/wealth*, *?wroth/wrath*, *?merry/mirth*
- *roomth*, *greenth*

-th

- *warmth*, *width*, *truth*, *depth*, ...
- Adj>N
- *heal/health*, *dead/death*, *young/youth*, *vile/filth*, *slow/sloth*
- *weal?/wealth*, *?wroth/wrath*, *?merry/mirth*
- *roomth*, *greenth*

Many enjoy the warmth, Vikings prefer the ***coolth***

Problem of Productivity

Problem of Productivity

- Which processes can be used to construct ***novel*** forms (e.g., -ness), which can only be ***reused*** in existing forms (e.g., -th)?

Problem of Productivity

- Which processes can be used to construct ***novel*** forms (e.g., -ness), which can only be ***reused*** in existing forms (e.g., -th)?
- How are such differences in productivity represented by the adult language user?

Problem of Productivity

- Which processes can be used to construct ***novel*** forms (e.g., -ness), which can only be ***reused*** in existing forms (e.g., -th)?
- How are such differences in productivity represented by the adult language user?
- How are such differences learned by the child?

Outline

Outline

1. The Proposal.

Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.

Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. English Derivational Morphology

Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. English Derivational Morphology
4. Conclusion

Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. English Derivational Morphology
4. Conclusion

The Proposal

The Proposal

1. Formalization of **what** can be reused.

The Proposal

1. Formalization of **what** can be reused.
   - Subcomputations.

The Proposal

1. Formalization of **what** can be reused.
   - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.

The Proposal

1. Formalization of **what** can be reused.
   - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
   - Optimal Bayesian inference.

The Proposal

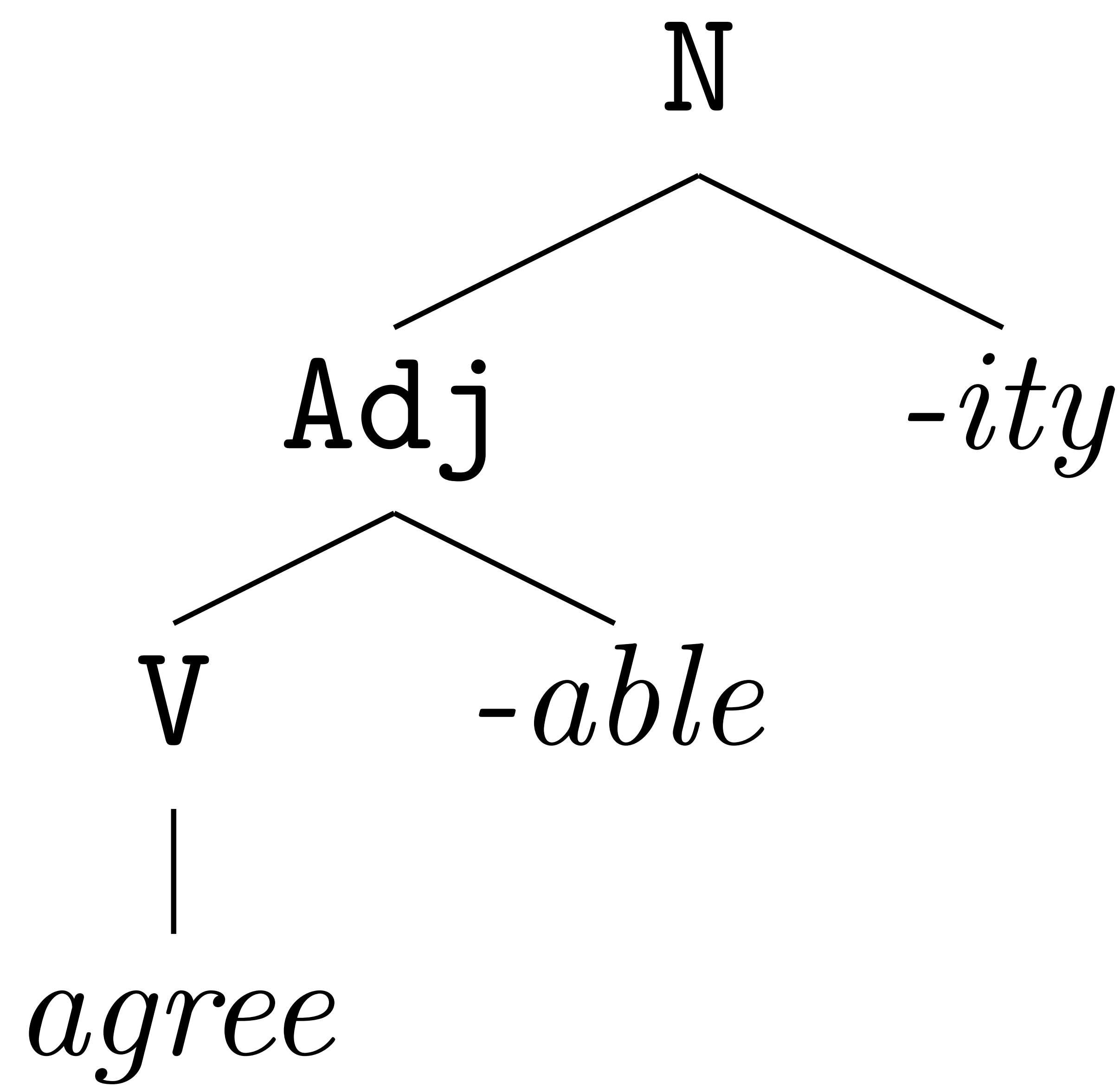
1. Formalization of **what** can be reused.
   - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
   - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

The Proposal

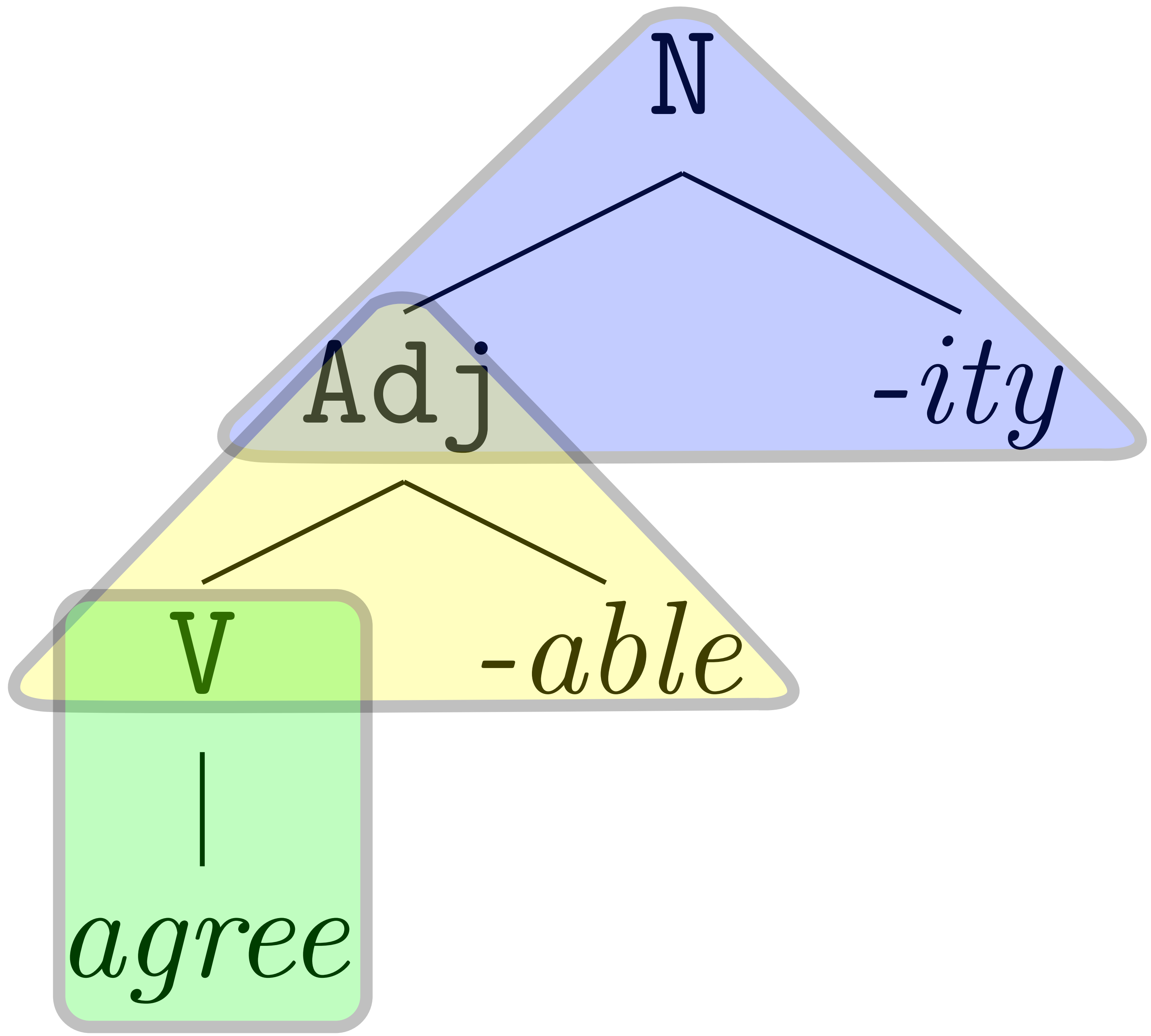
1. Formalization of **what** can be reused.
    - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
    - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

Starting Computational System

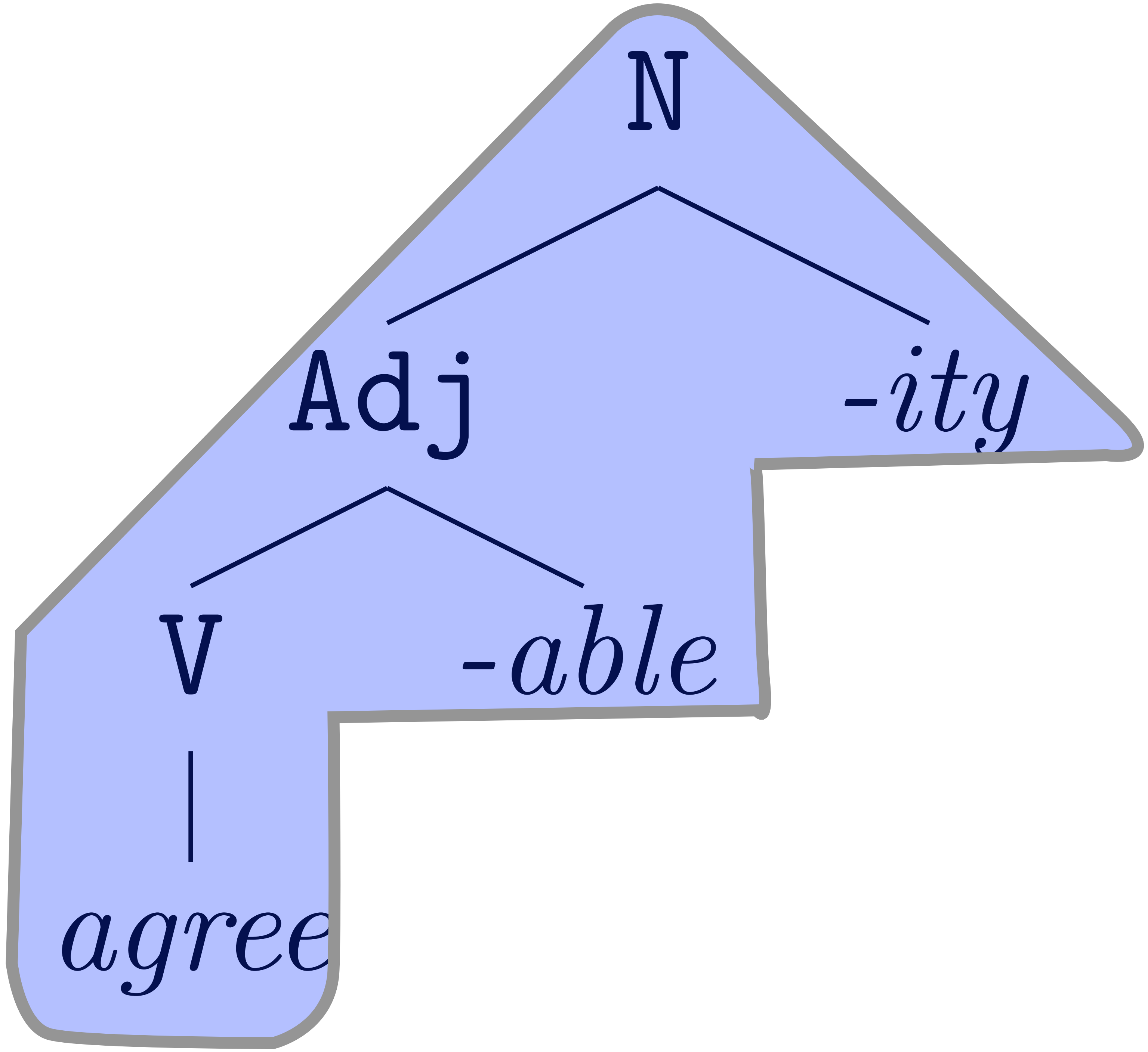
W	$\longrightarrow$	N	
W	$\longrightarrow$	V	
W	$\longrightarrow$	Adj	
W	$\longrightarrow$	Adv	
N	$\longrightarrow$	Adj	*-ness*
N	$\longrightarrow$	Adj	*-ity*
N	$\longrightarrow$	*electro-*	N
N	$\longrightarrow$	*magnet*	
N	$\longrightarrow$	*dog*	
...			
V	$\longrightarrow$	N	*-ify*
V	$\longrightarrow$	Adj	*-ize*
V	$\longrightarrow$	*re-*	V
V	$\longrightarrow$	*agree*	
V	$\longrightarrow$	*count*	
...			
Adj	$\longrightarrow$	*dis-*	Adj
Adj	$\longrightarrow$	V	*-able*
Adj	$\longrightarrow$	N	*-ic*
Adj	$\longrightarrow$	N	*-al*
Adj	$\longrightarrow$	*tall*	
...			
Adv	$\longrightarrow$	Adj	*-ly*
Adv	$\longrightarrow$	*today*	
...			



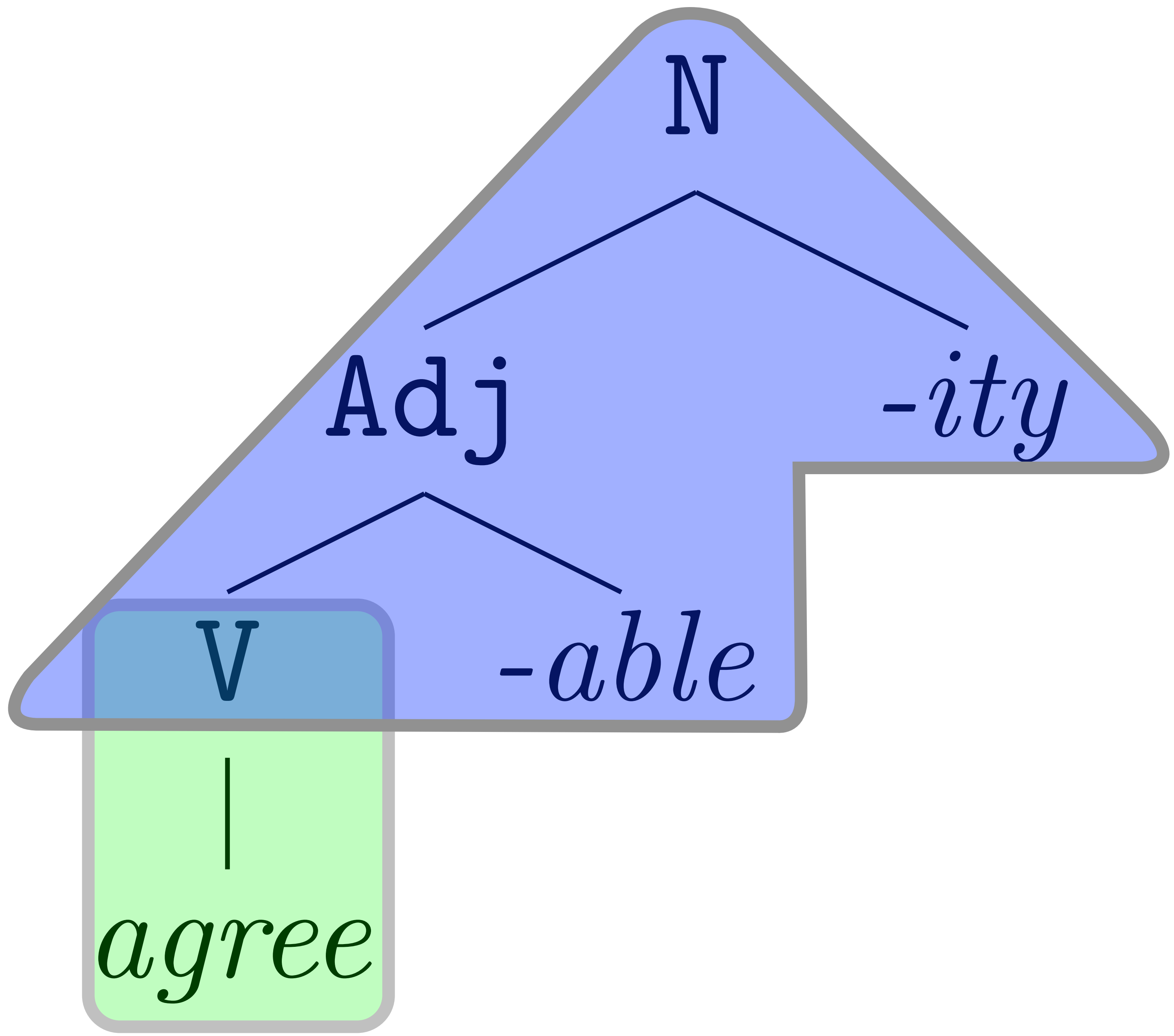
Subcomputations



Subcomputations



Subcomputations



The Proposal

1. Formalization of **what** can be reused.
    - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
    - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

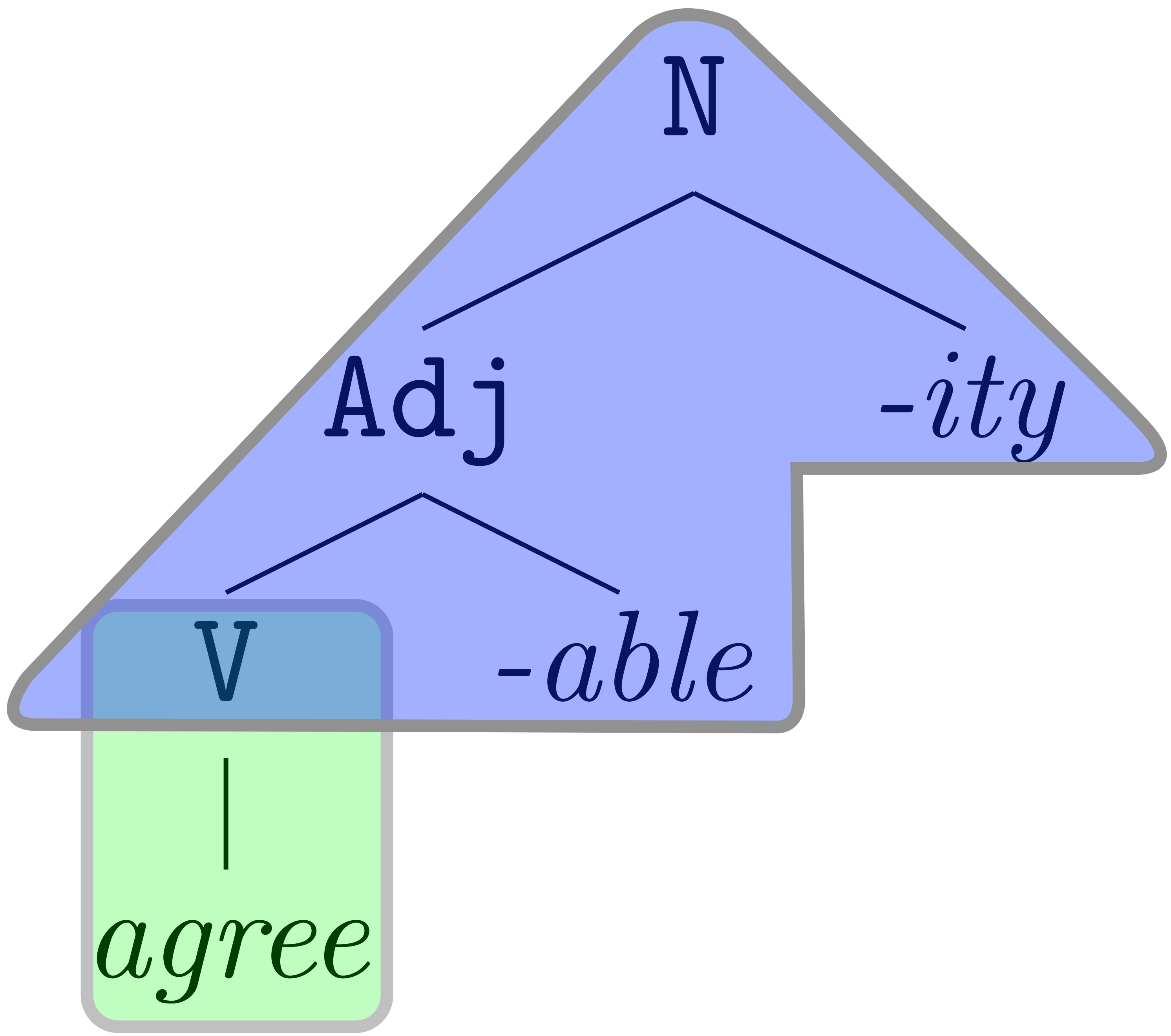
The Proposal

1. Formalization of **what** can be reused.
    - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
    - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

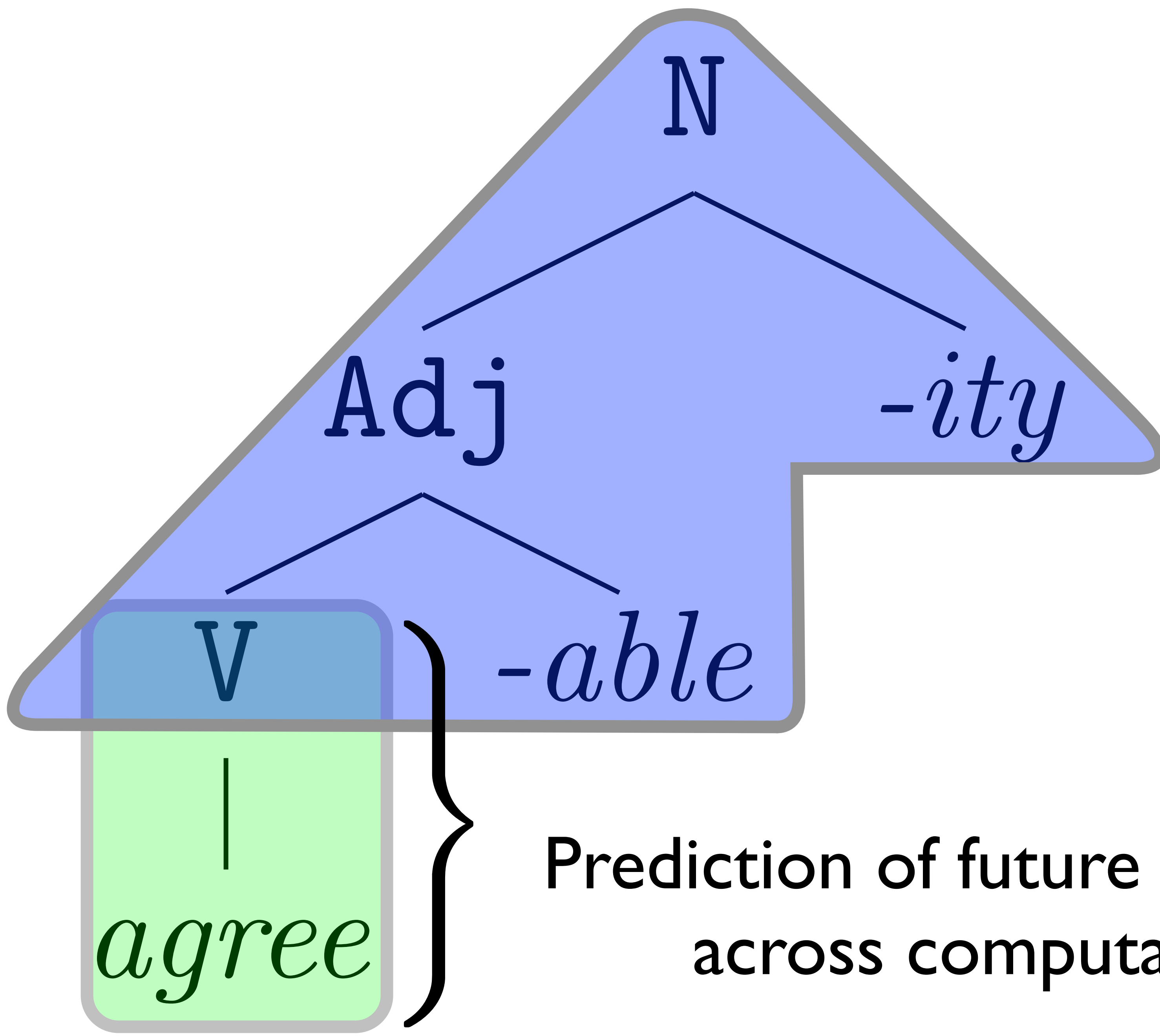
Bayesian Rational Analysis (Anderson, 1992)

- Find subcomputations which provide best explanation for the data.
- What *evidence* is available to the learner?
    - Which patterns give rise to productivity, which patterns imply reuse?

Subcomputations as Predictions

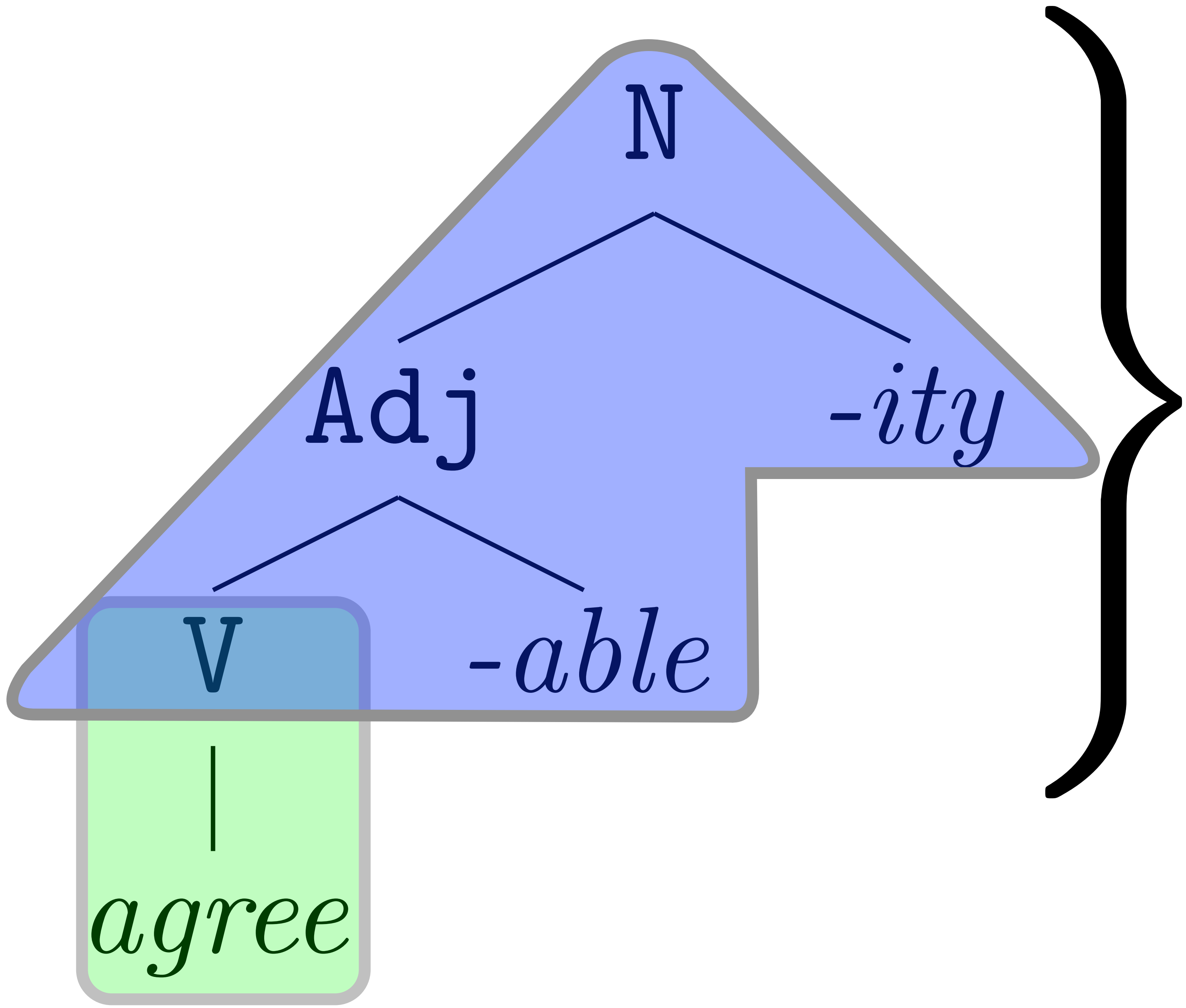


Subcomputations as Predictions



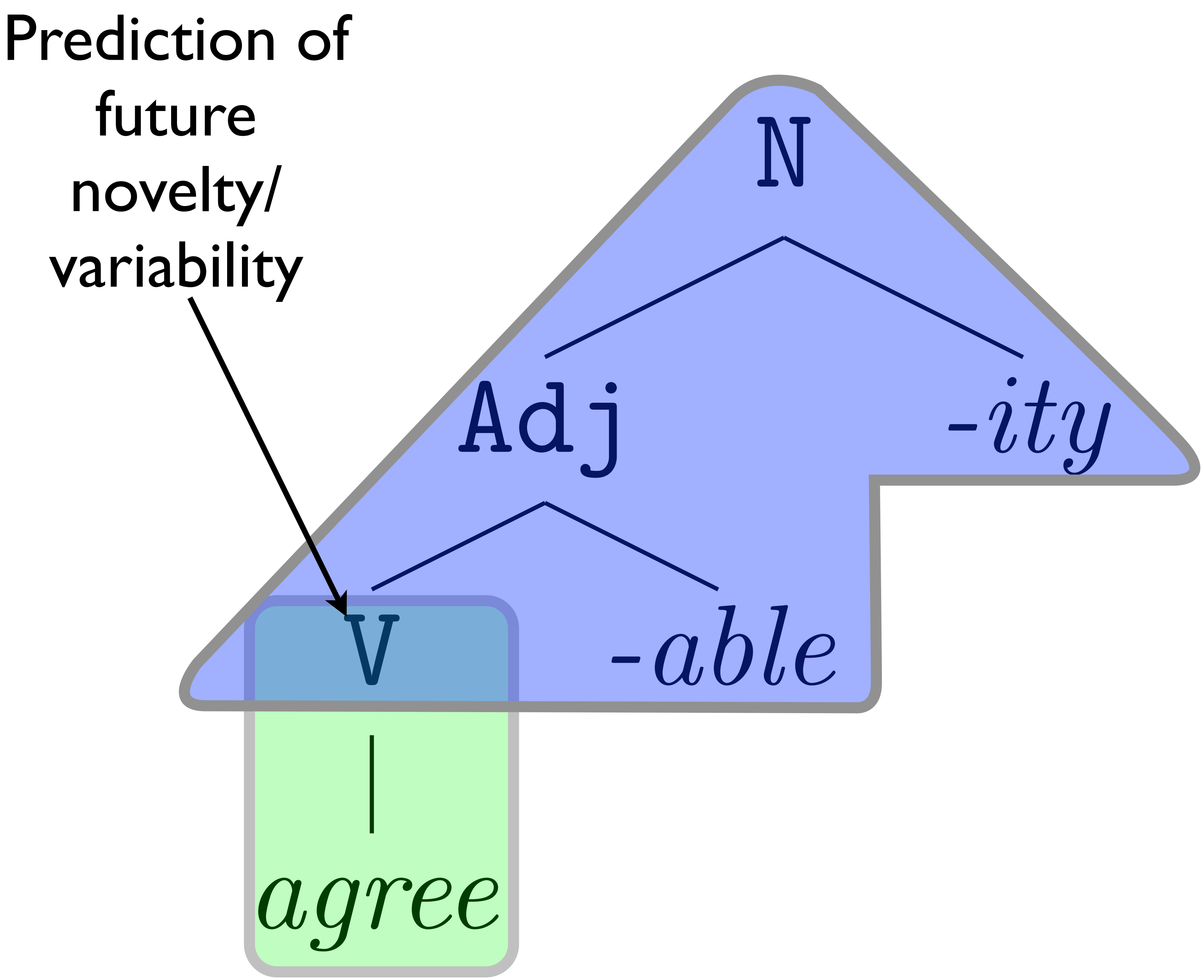
Prediction of future reusability
across computations

Subcomputations as Predictions

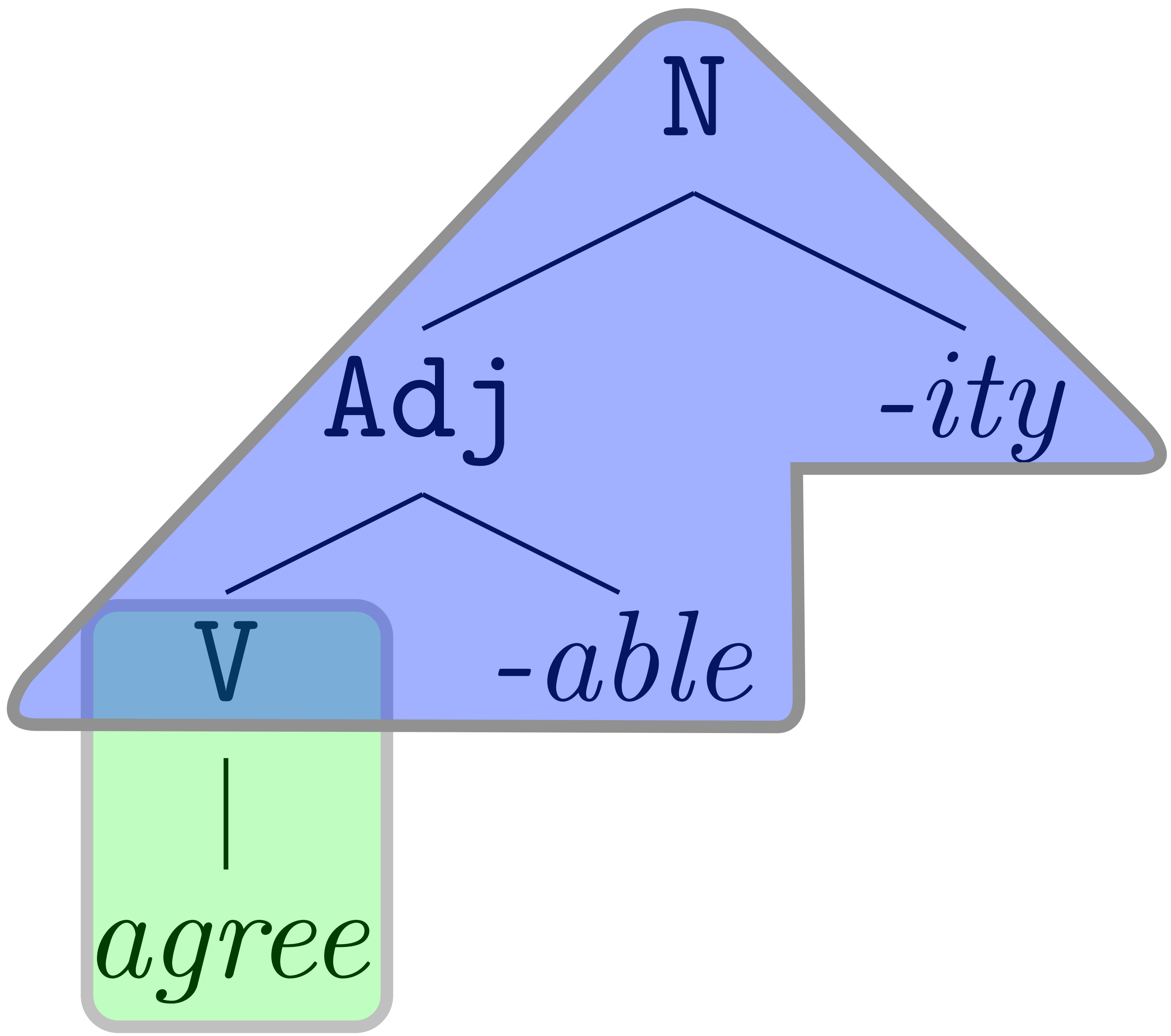


Prediction of future reusability of <u>*combination*</u>

Subcomputations as Predictions



Subcomputations as Predictions



Tradeoff between productivity and reuse

The Proposal

1. Formalization of **what** can be reused.
   - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
   - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

The Proposal

1. Formalization of **what** can be reused.
   - Subcomputations.
2. Formalization of **how** decision to reuse versus compute is made.
   - Optimal Bayesian inference.
3. The model from a probabilistic programming perspective.

The Formal Model:
Fragment Grammars

The Formal Model: *Fragment Grammars*

- Generalization of *Adaptor Grammars* (Johnson et al., 2007).

The Formal Model: *Fragment Grammars*

- Generalization of *Adaptor Grammars* (Johnson et al., 2007).
- Bayesian non-parametric distributions (*Pitman-Yor*).

The Formal Model: *Fragment Grammars*

- Generalization of *Adaptor Grammars* (Johnson et al., 2007).
- Bayesian non-parametric distributions (*Pitman-Yor*).
- Notion of *compiling* subcomputations via tools from probabilistic programming (Church language; Goodman et al., 2008).

The Formal Model: *Fragment Grammars*

- Generalization of *Adaptor Grammars* (Johnson et al., 2007).
- Bayesian non-parametric distributions (*Pitman-Yor*).
- Notion of *compiling* subcomputations via tools from probabilistic programming (Church language; Goodman et al., 2008).
  - Stochastic memoization (Johnson et al., 2007) of stochastically lazy/eager programs.

Fragment Grammars via Probabilistic Programming *(Church)*

Fragment Grammars via Probabilistic Programming *(Church)*

- Alternative to more standard mathematical formalization (see, O'Donnell, 2011).

Fragment Grammars via Probabilistic Programming *(Church)*

- Alternative to more standard mathematical formalization (see, O'Donnell, 2011).
- Highlights relationship between formalisms (PCFGs, Adaptor Grammars, Fragment Grammars).

Fragment Grammars via Probabilistic Programming *(Church)*

- Alternative to more standard mathematical formalization (see, O'Donnell, 2011).
- Highlights relationship between formalisms (PCFGs, Adaptor Grammars, Fragment Grammars).
- Cross fertilization of ideas from the theory of programming languages.

Fragment Grammars via Probabilistic Programming *(Church)*

- Alternative to more standard mathematical formalization (see, O'Donnell, 2011).
- Highlights relationship between formalisms (PCFGs, Adaptor Grammars, Fragment Grammars).
- Cross fertilization of ideas from the theory of programming languages.
- Caveat: Church inference algorithms do not work well for these models.

Fragment Grammars via Probabilistic Programming

Fragment Grammars via Probabilistic Programming

1. Stochastic computation via `unfold`

Fragment Grammars via Probabilistic Programming

1. Stochastic computation via `unfold`

2. Stochastic reuse via memoization

Fragment Grammars via Probabilistic Programming

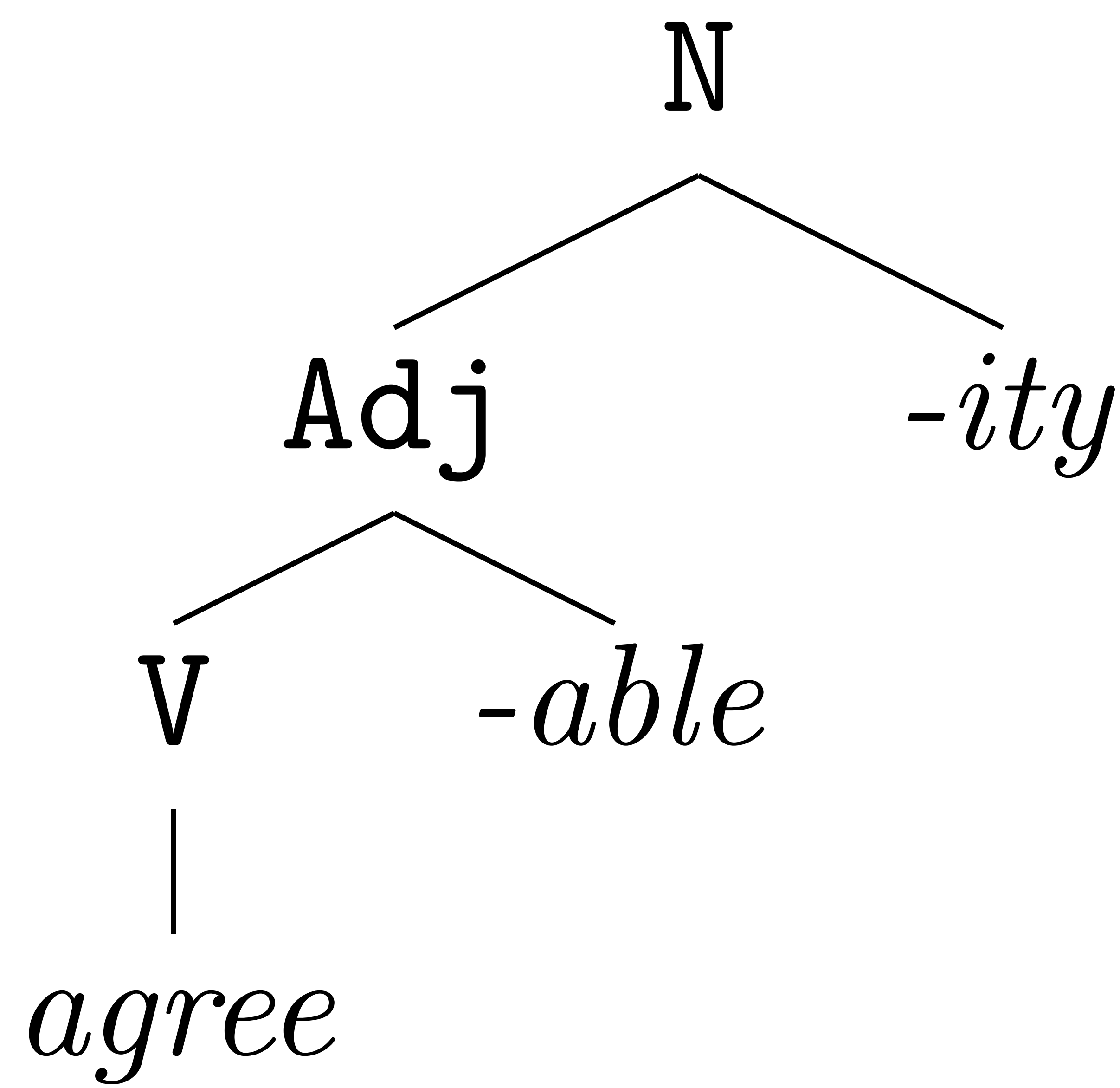
1. Stochastic computation via `unfold`
2. Stochastic reuse via memoization
3. Partial computations via stochastic laziness

Fragment Grammars via Probabilistic Programming

1. Stochastic computation via `unfold`
2. Stochastic reuse via memoization
3. Partial computations via stochastic laziness

Context Free Grammars

W	⟶	N	
W	⟶	V	
W	⟶	Adj	
W	⟶	Adv	
N	⟶	Adj	*-ness*
N	⟶	Adj	*-ity*
N	⟶	*electro-*	N
N	⟶	*magnet*	
N	⟶	*dog*	
...			
V	⟶	N	*-ify*
V	⟶	Adj	*-ize*
V	⟶	*re-*	V
V	⟶	*agree*	
V	⟶	*count*	
...			
Adj	⟶	*dis-*	Adj
Adj	⟶	V	*-able*
Adj	⟶	N	*-ic*
Adj	⟶	N	*-al*
Adj	⟶	*tall*	
...			
Adv	⟶	Adj	*-ly*
Adv	⟶	*today*	
...			



Declarative Knowledge of Constituent Structure

p_{W_1}	W	$\longrightarrow$	N	
p_{W_2}	W	$\longrightarrow$	V	
p_{W_3}	W	$\longrightarrow$	Adj	
p_{W_4}	W	$\longrightarrow$	Adv	
p_{N_1}	N	$\longrightarrow$	Adj	*-ness*
p_{N_2}	N	$\longrightarrow$	Adj	*-ity*
p_{N_3}	N	$\longrightarrow$	*electro-*	N
p_{N_4}	N	$\longrightarrow$	*magnet*	
p_{N_5}	N	$\longrightarrow$	*dog*	
	...			
p_{V_1}	V	$\longrightarrow$	N	*-ify*
p_{V_2}	V	$\longrightarrow$	Adj	*-ize*
p_{V_3}	V	$\longrightarrow$	*re-*	V
p_{V_4}	V	$\longrightarrow$	*agree*	
p_{V_5}	V	$\longrightarrow$	*count*	
	...			
p_{Adj_1}	Adj	$\longrightarrow$	*dis-*	Adj
p_{Adj_2}	Adj	$\longrightarrow$	V	*-able*
p_{Adj_3}	Adj	$\longrightarrow$	N	*-ic*
p_{Adj_4}	Adj	$\longrightarrow$	N	*-al*
p_{Adj_5}	Adj	$\longrightarrow$	*tall*	
	...			
p_{Adv_1}	Adv	$\longrightarrow$	Adj	*-ly*
p_{Adv_2}	Adv	$\longrightarrow$	*today*	
	...			

Declarative Knowledge of Constituent Structure

```
(define sample-rhs
  (lambda (nonterminal)
    (case nonterminal
      (('W) (multinomial (list (list 'N) (list 'V) (list 'Adj) (list 'Adv) ... )
                         (list p_W1 p_W2 p_W3 p_W4 ...)))
      (('N) (multinomial (list (list 'Adj 'ness) (list 'Adj 'ity) (list 'electro 'N) (list 'magnet) (list 'dog) ...)
                         (list p_N1 p_N2 p_N3 p_N4 p_N5 ...)))
      (('V) (multinomial (list (list 'N 'ify) (list 'Adj 'ize) (list 're 'V) (list 'agree) (list 'count) ...)
                         (list p_V1 p_V2 p_V3 p_V4 p_V5 ...)))
      (('Adj) (multinomial (list (list 'dis 'Adj) (list 'V 'able) (list 'N 'ic) (list 'N 'al) (list 'tall) ...)
                           (list p_Adj1 p_Adj2 p_Adj3 p_Adj4 p_Adj5 ...)))
      (('Adv) (multinomial (list (list 'Adj 'ly) (list 'today) ...)
                                 (list p_W1 p_W2 ...))))))
```

Fundamental Recursive Computation: `unfold`

```
(define unfold
 (lambda (symbol)
  (if (terminal? symbol)
   symbol
   (map unfold (sample-rhs symbol)))))
```

Fundamental Recursive Computation: `unfold`

```
(define unfold
 (lambda (symbol)
  (if (terminal? symbol)
   symbol
   (map unfold (sample-rhs symbol)))))
```

Choose a right-hand side for
symbol:
N → Adj *-ity*

Fundamental Recursive Computation: `unfold`

```
(define unfold
 (lambda (symbol)
  (if (terminal? symbol)
   symbol
   (map unfold (sample-rhs symbol)))))
```

Fundamental Recursive Computation: `unfold`

```
(define unfold
 (lambda (symbol)
  (if (terminal? symbol)
   symbol
   (map unfold (sample-rhs symbol)))))
```

Recursively apply `unfold` to each symbol on right-hand side

Computation Trace

```
(unfold 'N)
```

Computation Trace

```
(unfold ‘N)
```

```
(define unfold
   (lambda (symbol)
      (if (terminal? symbol)
         symbol
         (map unfold (sample-rhs symbol)))))
```

Computation Trace

(unfold ‘N)

(sample-rhs ‘N)

```
(define unfold
   (lambda (symbol)
      (if (terminal? symbol)
         symbol
         (map unfold (sample-rhs symbol)))))
```

Computation Trace

```
(unfold 'N)
     |
(sample-rhs 'N)
```

Computation Trace

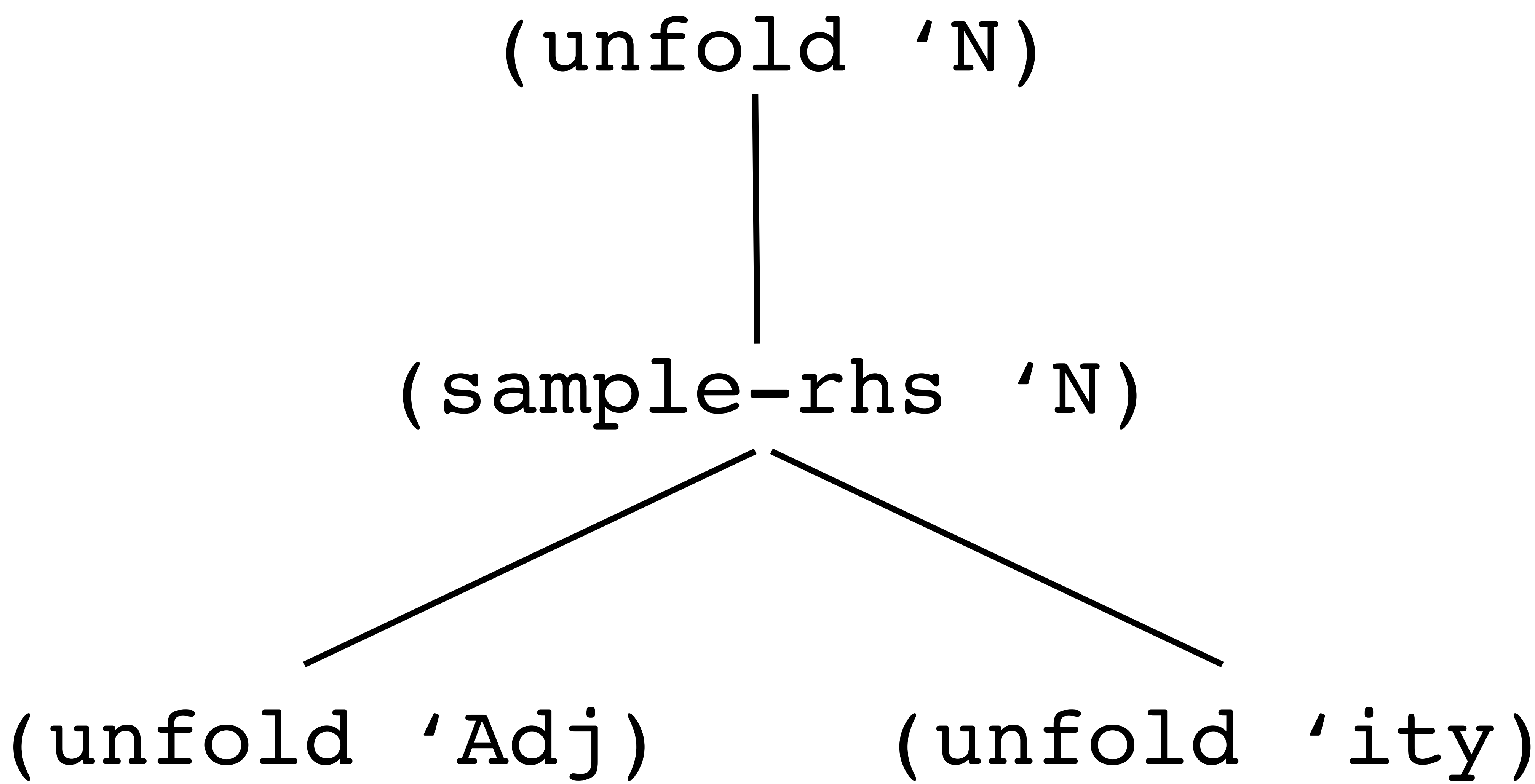
```
(unfold 'N)
     |
(sample-rhs 'N)
```

N → Adj *-ity*

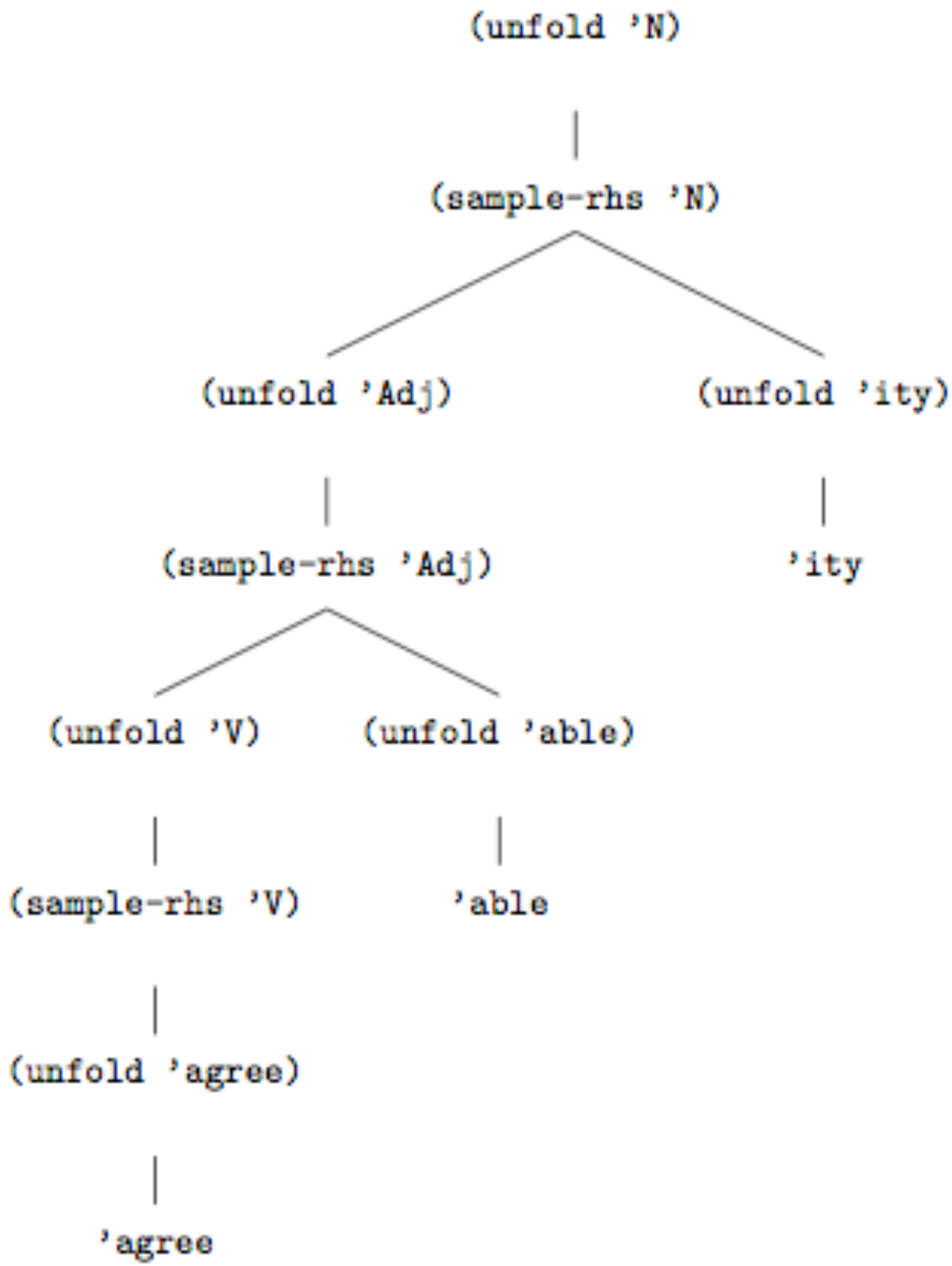
Computation Trace

```
(unfold 'N)
     |
(sample-rhs 'N)
```

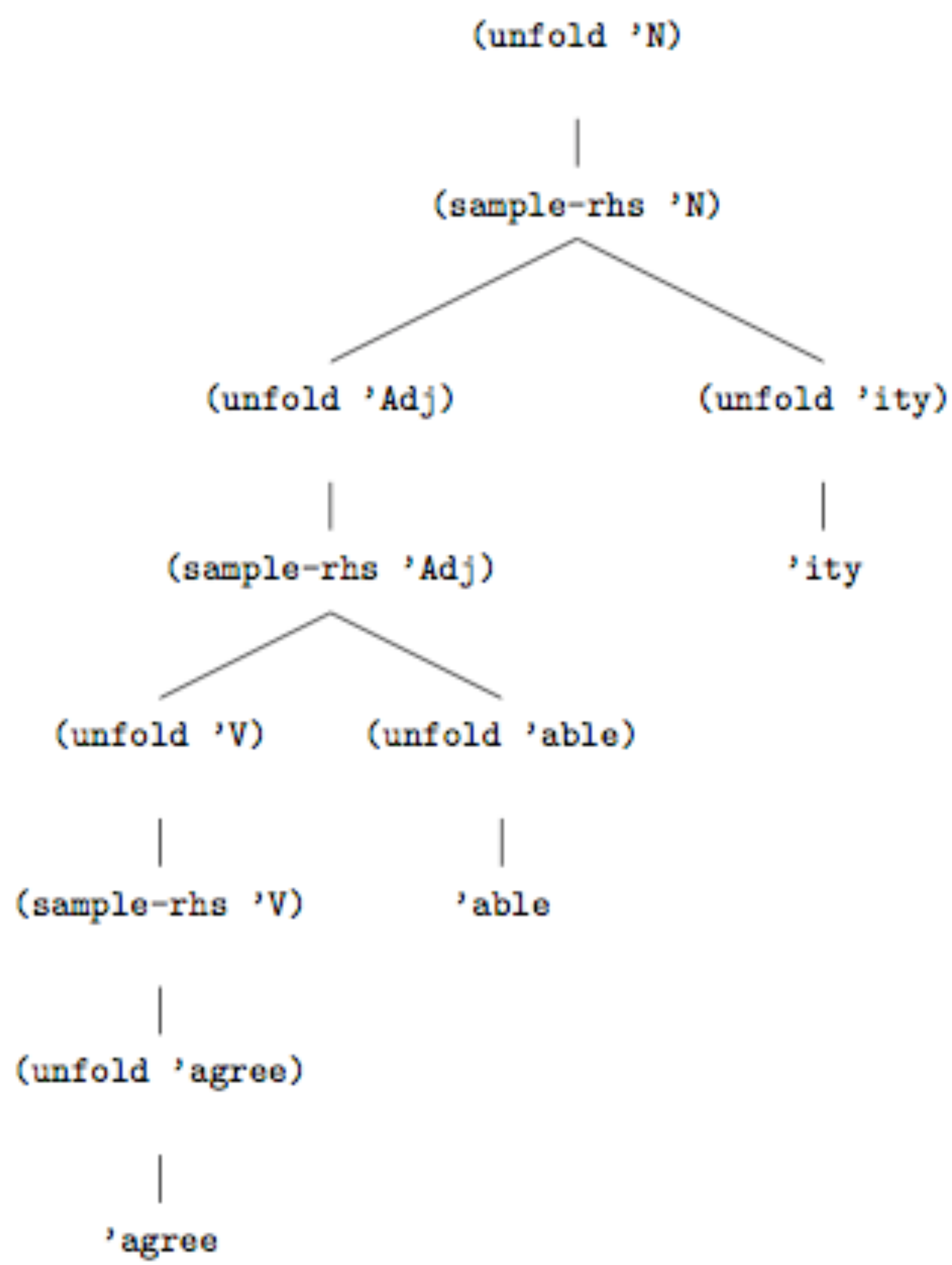
Computation Trace

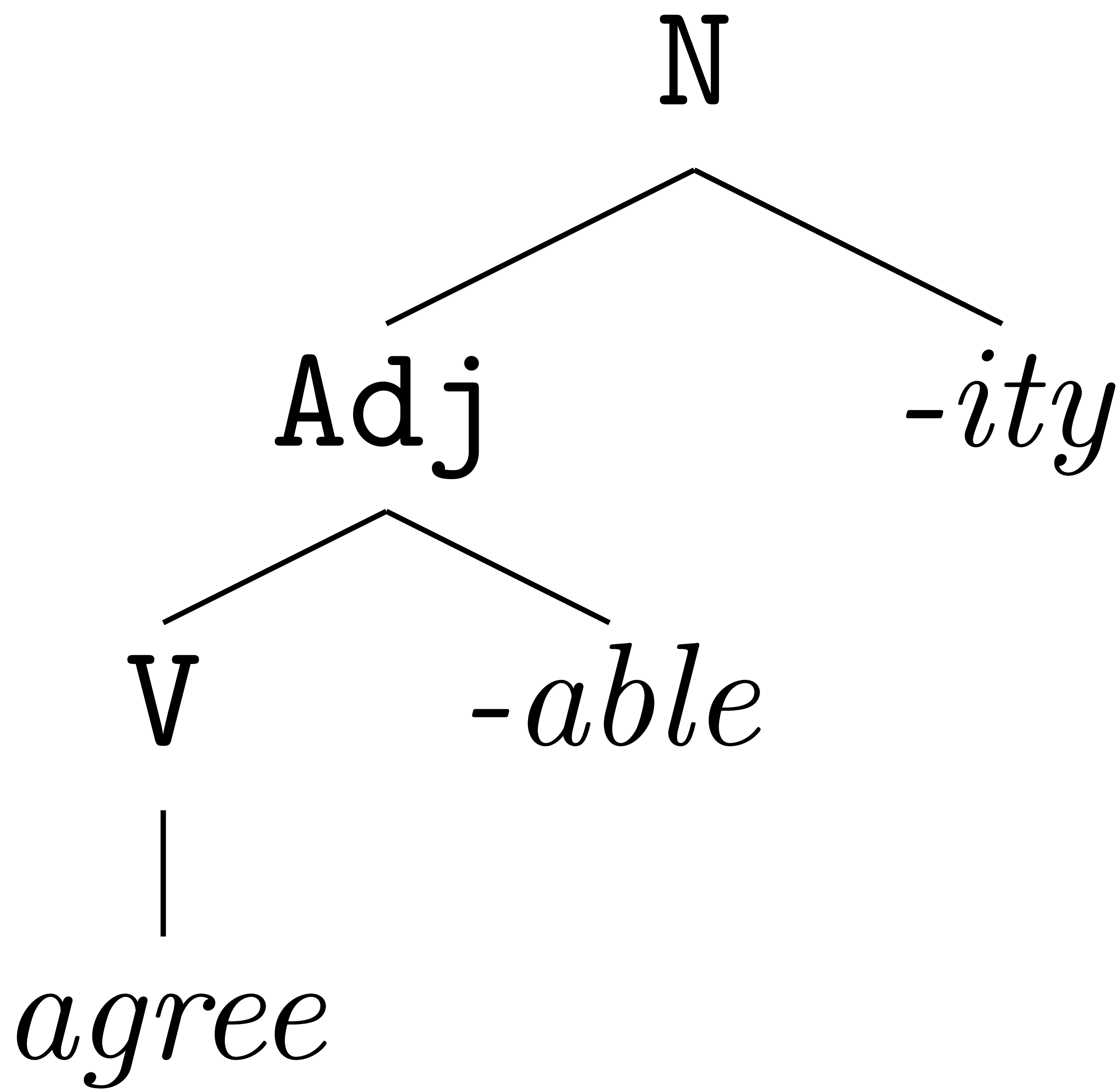


Computation Trace

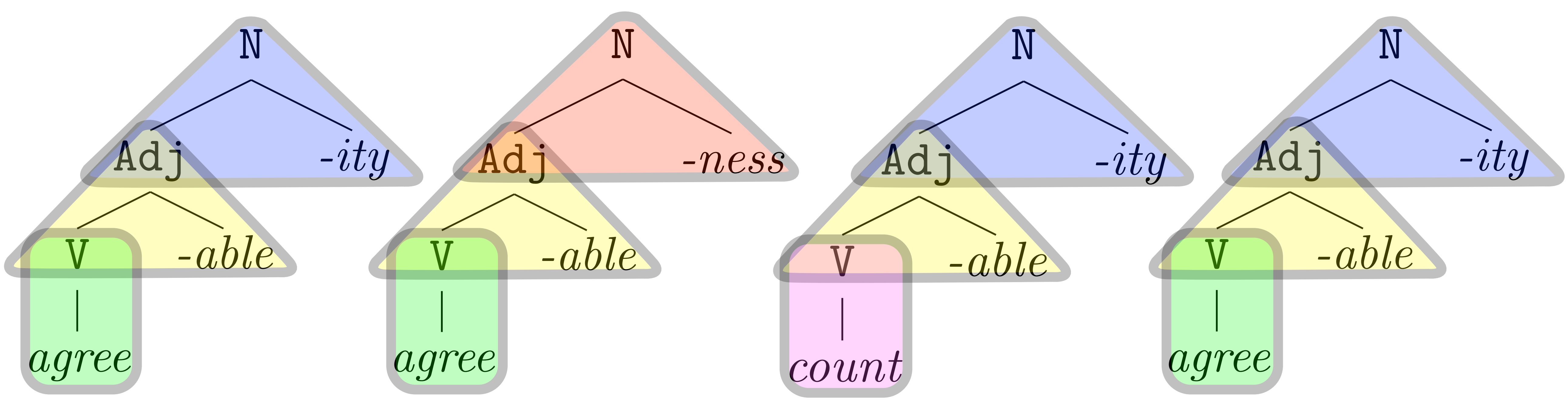


Trace as Tree





Reusability for PCFGs



Fragment Grammars via Probabilistic Programming

1. Stochastic computation via `unfold`
2. Stochastic reuse via memoization
3. Partial computations via stochastic laziness

Memoization

Memoization

- Store outputs of earlier computations in a table

Memoization

- Store outputs of earlier computations in a table
- When function is called with particular arguments then grab from table if stored

Memoization

- Store outputs of earlier computations in a table
- When function is called with particular arguments then grab from table if stored
- When function is called with new arguments, then compute and store in table

Memoization

- Store outputs of earlier computations in a table
- When function is called with particular arguments then grab from table if stored
- When function is called with new arguments, then compute and store in table
- Higher-order function: `mem`

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))
```

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))

    (eye-color ‘bob)  => ‘blue
```

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))
```

```
    (eye-color ‘bob)  => ‘blue
    (eye-color ‘bob)  => ‘brown
```

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))

    (eye-color ‘bob)  => ‘blue
    (eye-color ‘bob)  => ‘brown
    (eye-color ‘bob)  => ‘blue
```

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))

   (eye-color ‘bob)  => ‘blue
   (eye-color ‘bob)  => ‘brown
   (eye-color ‘bob)  => ‘blue
   (eye-color ‘bob)  => ‘brown
```

Reuse through Memoization

```
(define eye-color
  (lambda (person)
    (if (flip 0.5) ‘blue ‘brown)))

    (eye-color ‘bob)  => ‘blue
    (eye-color ‘bob)  => ‘brown
    (eye-color ‘bob)  => ‘blue
    (eye-color ‘bob)  => ‘brown
                         ...
```

Reuse through Memoization

```
(define eye-color
  (mem (lambda (person)
   (if (flip 0.5) 'blue brown))))
```

Reuse through Memoization

```
(define eye-color
  (mem (lambda (pers
    (if (flip 0.5) 'blue brown))))
```

Anywhere in the program where (**eye-color** 'bob) is used, we will *reuse* same value.

Reuse through Memoization

```
(define eye-color
  (mem (lambda (pers
    (if (flip 0.5) ‘blue brown))))
```

Anywhere in the program where (**eye-color** ‘bob) is used, we will *reuse* same value.

```
(eye-color ‘bob)  => ‘blue
```

Reuse through Memoization

```
(define eye-color
  (mem (lambda (pers
    (if (flip 0.5) ‘blue brown))))
```

Anywhere in the program where `(eye-color ‘bob)` is used, we will *reuse* same value.

```
(eye-color ‘bob)  => ‘blue
(eye-color ‘bob)  => ‘blue
```

Reuse through Memoization

```
(define eye-color
  (mem (lambda (pers
   (if (flip 0.5) 'blue brown))))
```

Anywhere in the program where (**eye-color** 'bob) is used, we will *reuse* same value.

```
(eye-color 'bob)  => 'blue
(eye-color 'bob)  => 'blue
(eye-color 'bob)  => 'blue
```

Reuse through Memoization

Anywhere in the program where (**eye-color** ‘bob) is used, we will *reuse* same value.

```
(define eye-color
  (mem (lambda (pers
    (if (flip 0.5) ‘blue brown))))
```

```
(eye-color ‘bob)   =>  ‘blue
(eye-color ‘bob)   =>  ‘blue
(eye-color ‘bob)   =>  ‘blue
(eye-color ‘bob)   =>  ‘blue
```

Reuse through Memoization

Anywhere in the program where (**eye-color** 'bob) is used, we will *reuse* same value.

```
(define eye-color
  (mem (lambda (pers
    (if (flip 0.5) 'blue brown))))
```

```
(eye-color 'bob)   => 'blue
(eye-color 'bob)   => 'blue
(eye-color 'bob)   => 'blue
(eye-color 'bob)   => 'blue
                   ...
```

Stochastic Reusability

- Deterministic memoization always returns same value after first call, but sometimes we want to **probabilistically** favor reuse.

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))
```

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))


(location 'bob)  => 'UCLA
```

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))


(location ‘bob)  => ‘UCLA
(location ‘bob)  => ‘Antarctica
```

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))
```

```
(location 'bob)  => 'UCLA
(location 'bob)  => 'Antarctica
(location 'bob)  => 'London
```

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))


(location 'bob)  => 'UCLA
(location 'bob)  => 'Antarctica
(location 'bob)  => 'London
(location 'bob)  => 'Thailand
```

Stochastic Reusability

```
(define location
  (lambda (person)
   (sample-location-in-world)))


(location 'bob)  => 'UCLA
(location 'bob)  => 'Antarctica
(location 'bob)  => 'London
(location 'bob)  => 'Thailand
                ...
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))


 (location 'bob)  => 'home
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))
```

```
(location 'bob)  => 'home
(location 'bob)  => 'office
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))
```

```
(location 'bob)  => 'home
(location 'bob)  => 'office
(location 'bob)  => 'home
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))
```

```
(location 'bob)  => 'home
(location 'bob)  => 'office
(location 'bob)  => 'home
(location 'bob)  => 'home
```

Stochastic Reusability

```
(define location
  (stochastic-mem (lambda (person)
   (sample-location-in-world))))
```

```
(location 'bob)  => 'home
(location 'bob)  => 'office
(location 'bob)  => 'home
(location 'bob)  => 'home
                 ...
```

Stochastic Memoization

(Goodman et al., 2008; Johnson et al., 2007)

Stochastic Memoization

(Goodman et al., 2008; Johnson et al., 2007)

- Adaptor Grammars: Anything that can be computed can be stored and reused *probabilistically*.

Stochastic Memoization

(Goodman et al., 2008; Johnson et al., 2007)

- Adaptor Grammars: Anything that can be computed can be stored and reused *probabilistically*.
- Memoization distribution: *Pitman-Yor Processes* (Pitman & Yor, 1995).

Stochastic Memoization

(Goodman et al., 2008; Johnson et al., 2007)

- Adaptor Grammars: Anything that can be computed can be stored and reused *probabilistically*.
- Memoization distribution: *Pitman-Yor Processes* (Pitman & Yor, 1995).
- Stochastic memoization + PCFGs = Adaptor Grammars.

Pitman-Yor Process

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - $a \in [0,1]$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - $a \in [0,1]$
  - $b > -a$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - $a \in [0,1]$
  - $b > -a$

Probability of Reuse

$$\frac{y_i - a}{N + b}$$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - a $\in [0,1]$
  - b > -a

y_i: Total number of observations of value i

Probability of Reuse

$$\frac{y_i - a}{N + b}$$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - a $\in [0,1]$
  - b > -a

y_i: Total number of observations of value i

N: Total number of observations

Probability of Reuse

$$\frac{y_i - a}{N + b}$$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - $a \in [0,1]$
  - $b > -a$

y_i: Total number of observations of value i

N: Total number of observations

Probability of Reuse

$$\frac{y_i - a}{N + b}$$

Probability of Novelty

$$\frac{a \cdot K + b}{N + b}$$

Pitman-Yor Process

- Generalization of the Chinese Restaurant Process
- Two parameters:
  - a $\in [0,1]$
  - b > -a

y_i: Total number of observations of value i

N: Total number of observations

K: Total number of values

Probability of Reuse

$$\frac{y_i - a}{N + b}$$

Probability of Novelty

$$\frac{a \cdot K + b}{N + b}$$

Properties of PYPs

Properties of PYPs

- Rich get richer, concentrates distribution on a few values.

Properties of PYPs

- Rich get richer, concentrates distribution on a few values.
- Prefers fewer customers/tables/tables-per-customer.

Properties of PYPs

- Rich get richer, concentrates distribution on a few values.
- Prefers fewer customers/tables/tables-per-customer.
- Prefers to generate novel values proportional to how often novelty has been generated in the past.

Adaptor Grammars

(Johnson et al., 2007)

```
(define adapted-unfold
  (PYMem a b
    (lambda (symbol)
      (if (terminal? symbol)
        symbol
        (map unfold (sample-rhs symbol))))))
```

Properties of Adaptor Grammars

Properties of Adaptor Grammars

- Reuse previous computations (subtrees).

Properties of Adaptor Grammars

- Reuse previous computations (subtrees).
- Can compute novel items productively using base system.

Properties of Adaptor Grammars

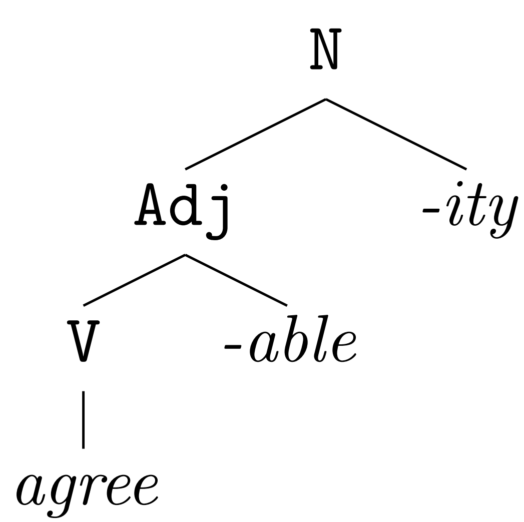
- Reuse previous computations (subtrees).
- Can compute novel items productively using base system.
- Build new stored trees recursively.

Properties of Adaptor Grammars

- Reuse previous computations (subtrees).
- Can compute novel items productively using base system.
- Build new stored trees recursively.
- Only reuse complete subtrees (on adapted nonterminals).

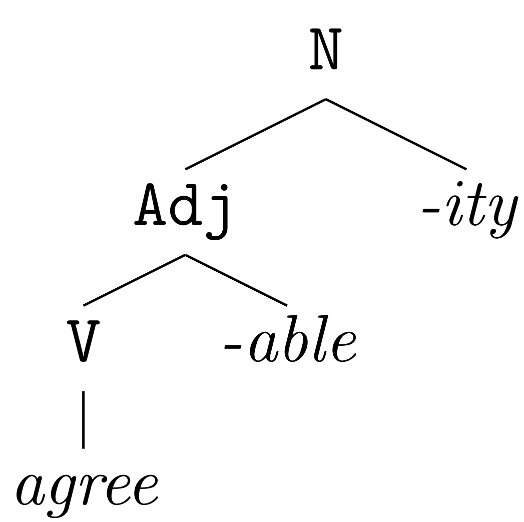
Properties of Adaptor Grammars

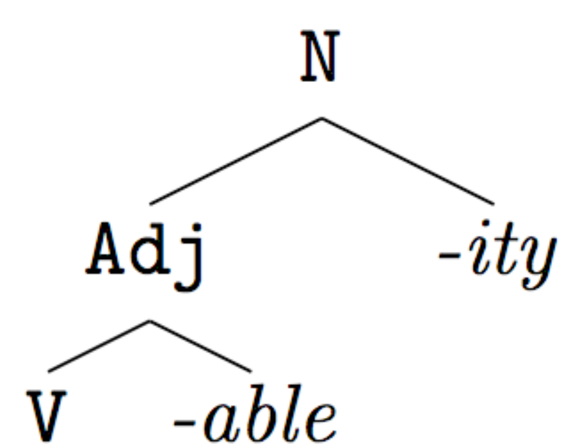
- Reuse previous computations (subtrees).
- Can compute novel items productively using base system.
- Build new stored trees recursively.
- Only reuse complete subtrees (on adapted nonterminals).



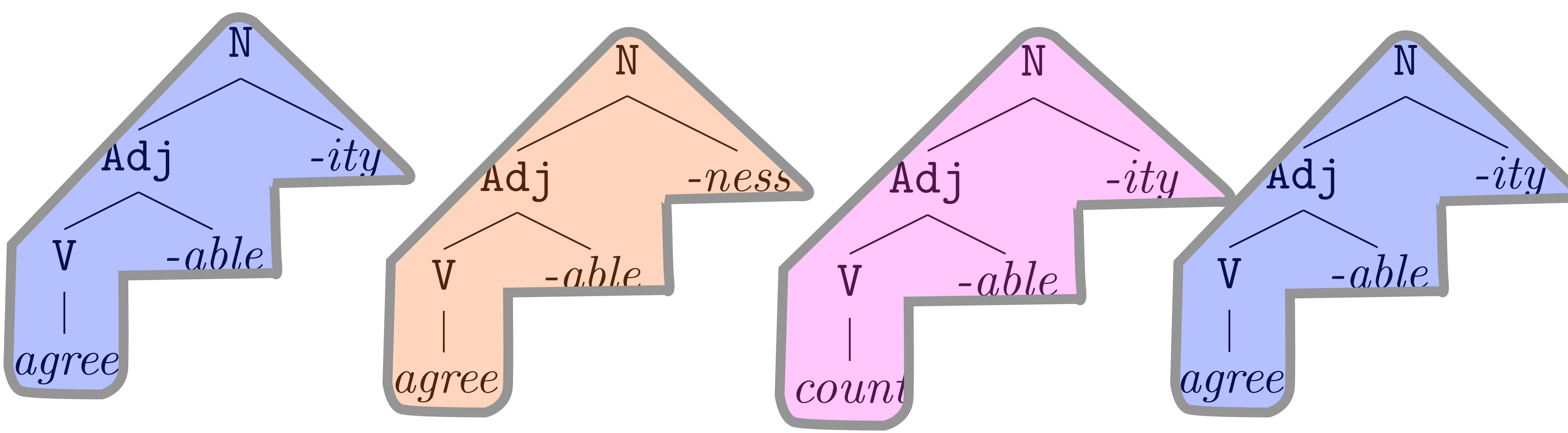
Properties of Adaptor Grammars

- Reuse previous computations (subtrees).
- Can compute novel items productively using base system.
- Build new stored trees recursively.
- Only reuse complete subtrees (on adapted nonterminals).



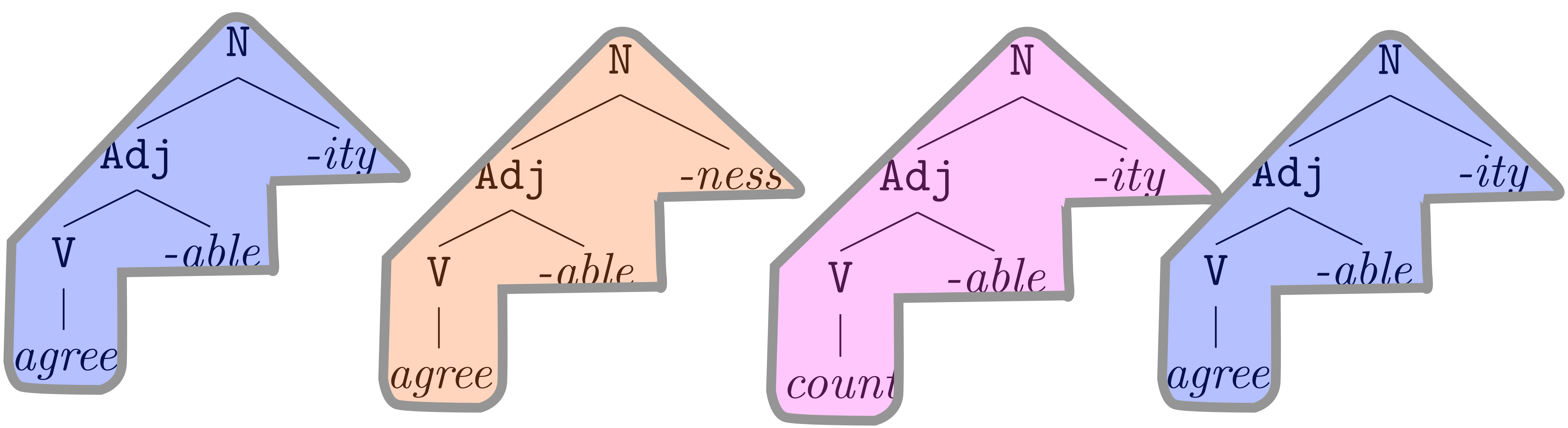


Reusability for Adaptor Grammars



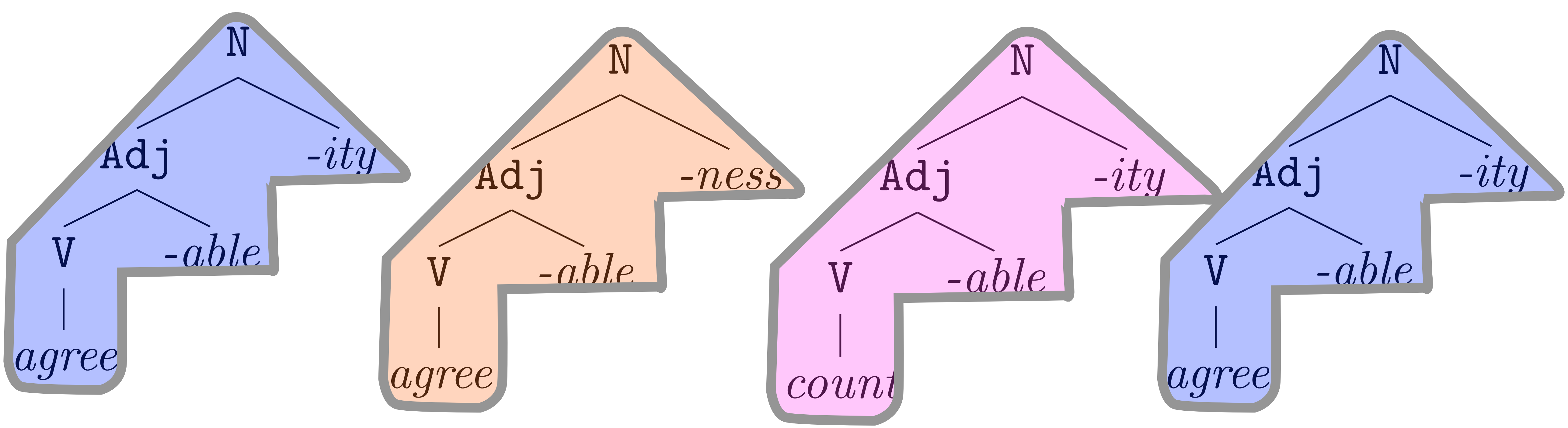
Reusability for Adaptor Grammars

1. Always possible to use base grammar.



Reusability for Adaptor Grammars

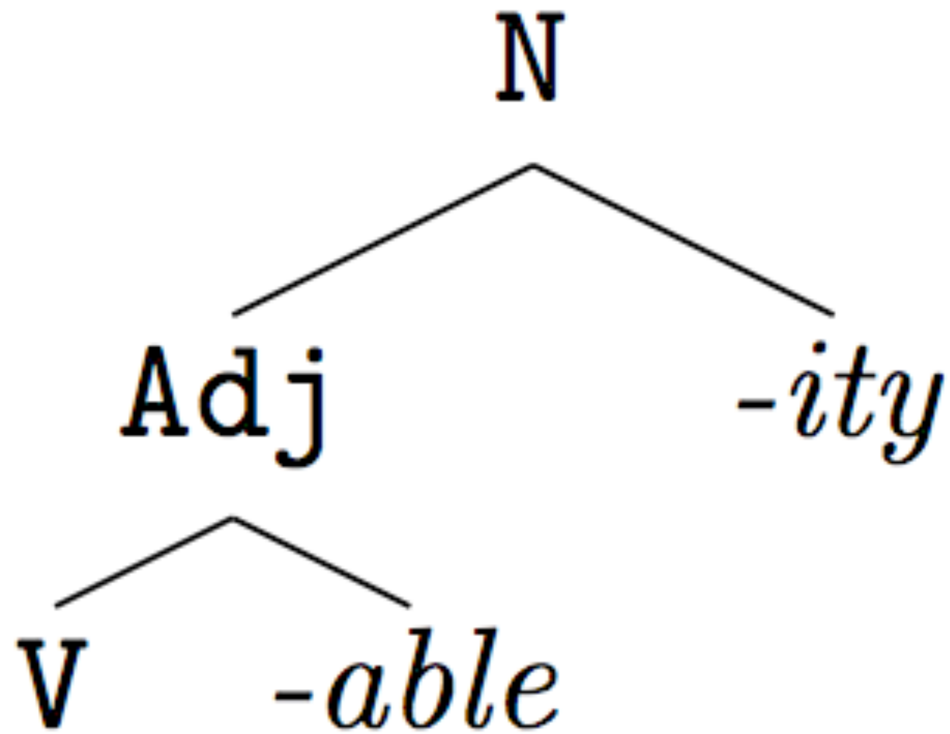
1. Always possible to use base grammar.
2. Fully recursive.



Fragment Grammars via Probabilistic Programming

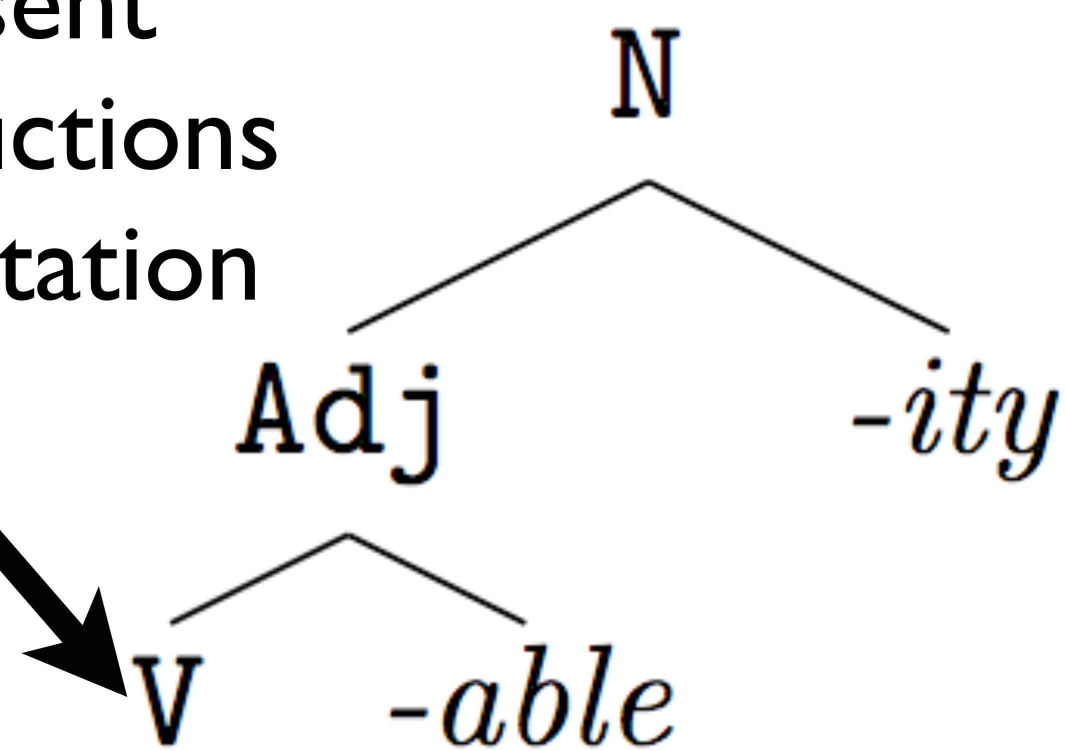
1. Stochastic computation via `unfold`
2. Stochastic reuse via memoization
3. Partial computations via stochastic laziness

Goal: Represent Partial Computations



Goal: Represent Partial Computations

Variables represent "delayed" instructions for later computation



Lazy and Eager Evaluation

Lazy and Eager Evaluation

- Eager Evaluation: Do as much work as early as possible.

Lazy and Eager Evaluation

- Eager Evaluation: Do as much work as early as possible.
- Lazy Evaluation: Delay work until it is absolutely necessary to continue computation.

Example

```
(define add3
 (lambda (x y z)
   (+ x y z)))
```

Eager Evaluation

```
(add3 (+ 1 2 3) (* 2 4) (- 3 1))
```

Eager Evaluation

```
(add3 (+ 1 2 3) (* 2 4) (- 3 1))
```

Eager Evaluation

```
(add3 6 (* 2 4) (- 3 1))
```

Eager Evaluation

```
(add3 6 (* 2 4) (- 3 1))
```

Eager Evaluation

```
(add3 6 8 (- 3 1))
```

Eager Evaluation

```
(add3 6 8 (- 3 1))
```

Eager Evaluation

```
(add3 6 8 2)
```

Eager Evaluation

```
(define add3
  (lambda (x y z)
    (+ x y z)))
```

```
(add3 6 8 2)
```

Eager Evaluation

```
(define add3

  (lambda (x y z)

    (+ x y z)))
```

```
(+ 6 8 2)
   x y z
```

Eager Evaluation

```
16
```

Lazy Evaluation

```
(add3 (+ 1 2 3) (* 2 4) (- 3 1))
```

Lazy Evaluation

```
(define add3
  (lambda (x y z)
    (+ x y z)))
```

```
(add3 (+ 1 2 3) (* 2 4) (- 3 1))
```

Lazy Evaluation

```
(define add3
  (lambda (x y z)
    (+ x y z)))
```

```
(+ (+ 1 2 3) (* 2 4) (- 3 1))
```

x y z

Lazy Evaluation

```
(define add3
  (lambda (x y z)
    (+ x y z)))
```

```
(+ (+ 1 2 3) (* 2 4) (- 3 1))
   x         y       z
```

Argument expressions are delayed until their values are needed by another computation.

Lazy Evaluation

```
(+ (+ 1 2 3) (* 2 4) (- 3 1))
```

Primitive + procedure forces evaluation of arguments.

Lazy Evaluation

```
(+ (+ 1 2 3) (* 2 4) (- 3 1))
```

Lazy Evaluation

```
(+ 16 (* 2 4) (- 3 1))
```

Lazy Evaluation

```
(+ 16 (* 2 4) (- 3 1))
```

Lazy Evaluation

```
(+ 16 8 (- 3 1))
```

Lazy Evaluation

```
(+ 16 8 (- 3 1))
```

Lazy Evaluation

```
(+ 16 8 2)
```

Lazy Evaluation

```
16
```

λ-calculus: Order of Evaluation

λ-calculus: Order of Evaluation

- *Applicative order* (eager evaluation): evaluate arguments first, then apply function.

λ-calculus: Order of Evaluation

- *Applicative order* (eager evaluation): evaluate arguments first, then apply function.
- *Normal order* (lazy evaluation): copy arguments into procedure, only evaluate when needed.

λ-calculus: Order of Evaluation

- *Applicative order* (eager evaluation): evaluate arguments first, then apply function.
- *Normal order* (lazy evaluation): copy arguments into procedure, only evaluate when needed.
- *Church-Rosser theorem*: Order doesn't matter for deterministic λ-calculus.

λ-calculus: Order of Evaluation

- *Applicative order* (eager evaluation): evaluate arguments first, then apply function.
- *Normal order* (lazy evaluation): copy arguments into procedure, only evaluate when needed.
- *Church-Rosser theorem*: Order doesn't matter for deterministic λ-calculus.
- Does matter for Ψλ-calculus!

Ψλ-calculus: Order of Evaluation

```
(define same?
  (lambda (x)
    (equal? x x)))
```

Ψλ-calculus: Order of Evaluation

```
(define same?
 (lambda (x)
   (equal? x x)))
```

Ψλ-calculus: Order of Evaluation

```
(define same?
  (lambda (x)
    (equal? x x)))
```

```
(same? (flip))
```

Ψλ-calculus: Order of Evaluation

```
(define same?
  (lambda (x)
    (equal? x x)))
```

`(same? (flip))` —eager→ P(`true`) = 1

Ψλ-calculus: Order of Evaluation

```
(define same?
  (lambda (x)
    (equal? x x)))
```

`(same? (flip))`

- eager → P(true) = 1
- lazy → P(true) = 1/2

Tradeoff

- Laziness allows you to delay computation and, thus, **preserve randomness** and variability until the last possible moment.
- Eagerness allows you to determine random choices early in computation and, thus, **share** choices across different parts of a program.

Random Evaluation Order

Random Evaluation Order

- Idea: Stochastically mix lazy and eager evaluation in Ψλ-calculus.

Random Evaluation Order

- Idea: Stochastically mix lazy and eager evaluation in $\Psi\lambda$-calculus.
- Ultimately allow **learning** of which computations should be performed in advance and which should be delayed.

Random Evaluation Order

- Idea: Stochastically mix lazy and eager evaluation in Ψλ-calculus.
- Ultimately allow **learning** of which computations should be performed in advance and which should be delayed.
- Assume eager evaluation strategy and add `delay` primitive.

Random Evaluation Order

- Idea: Stochastically mix lazy and eager evaluation in Ψλ-calculus.
- Ultimately allow **learning** of which computations should be performed in advance and which should be delayed.
- Assume eager evaluation strategy and add `delay` primitive.
- Apply to `unfold` (can be applied fully generally).

Stochastic Lazy `unfold`

```
(define stochastic-lazy-unfold
  (lambda (symbol)
    (if (terminal? symbol)
      symbol
      (map delay-or-unfold (sample-rhs symbol)))))
```

Stochastic Lazy `unfold`

```
(define stochastic-lazy-unfold
 (lambda (symbol)
   (if (terminal? symbol)
     symbol
     (map delay-or-unfold (sample-rhs symbol)))))
```

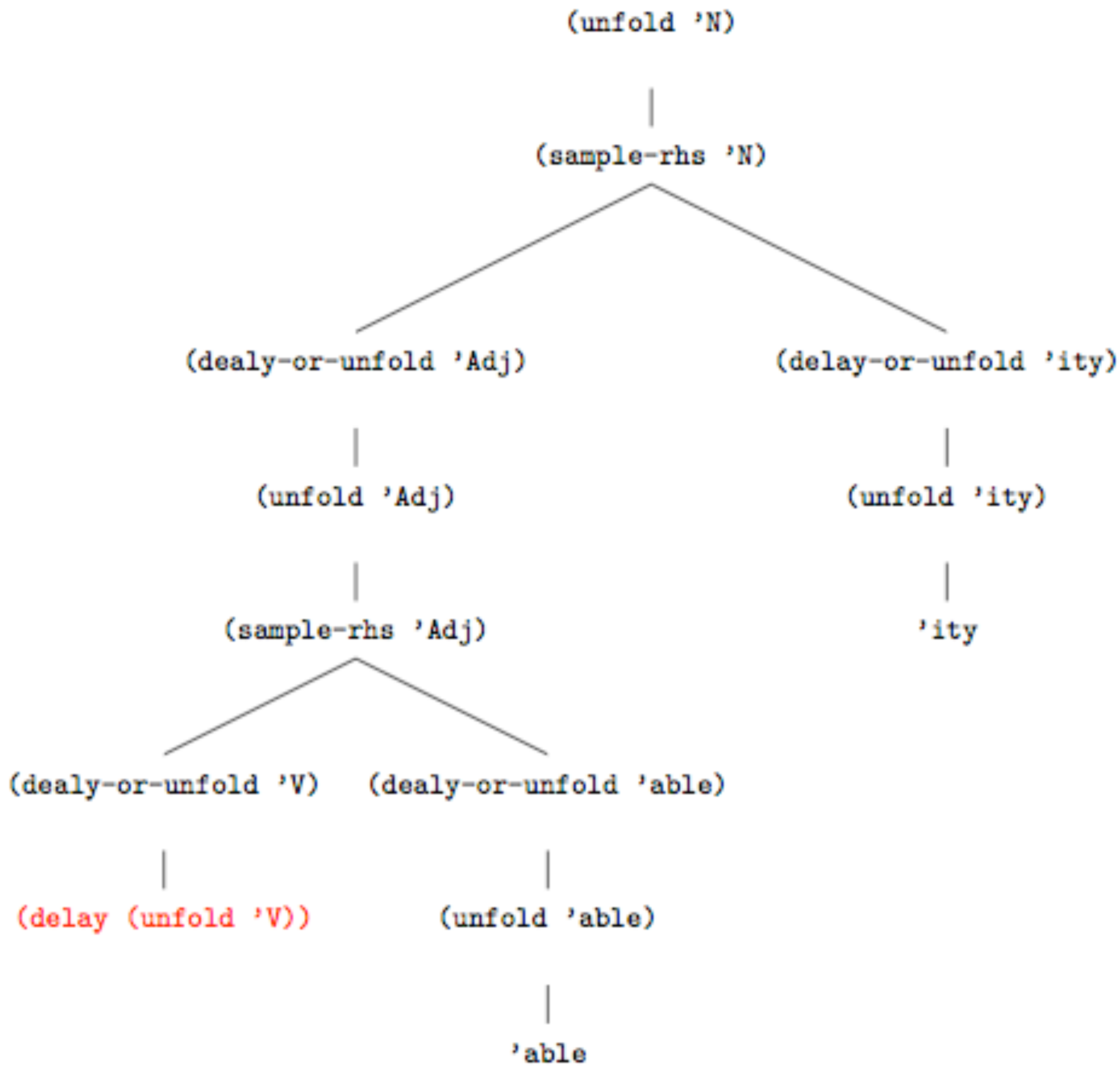
Stochastic Lazy `unfold`

```
(define delay-or-unfold
  (lambda (symbol)
    (if (flip)
      (delay (stochastic-lazy-unfold symbol))
      (stochastic-lazy-unfold symbol))))
```

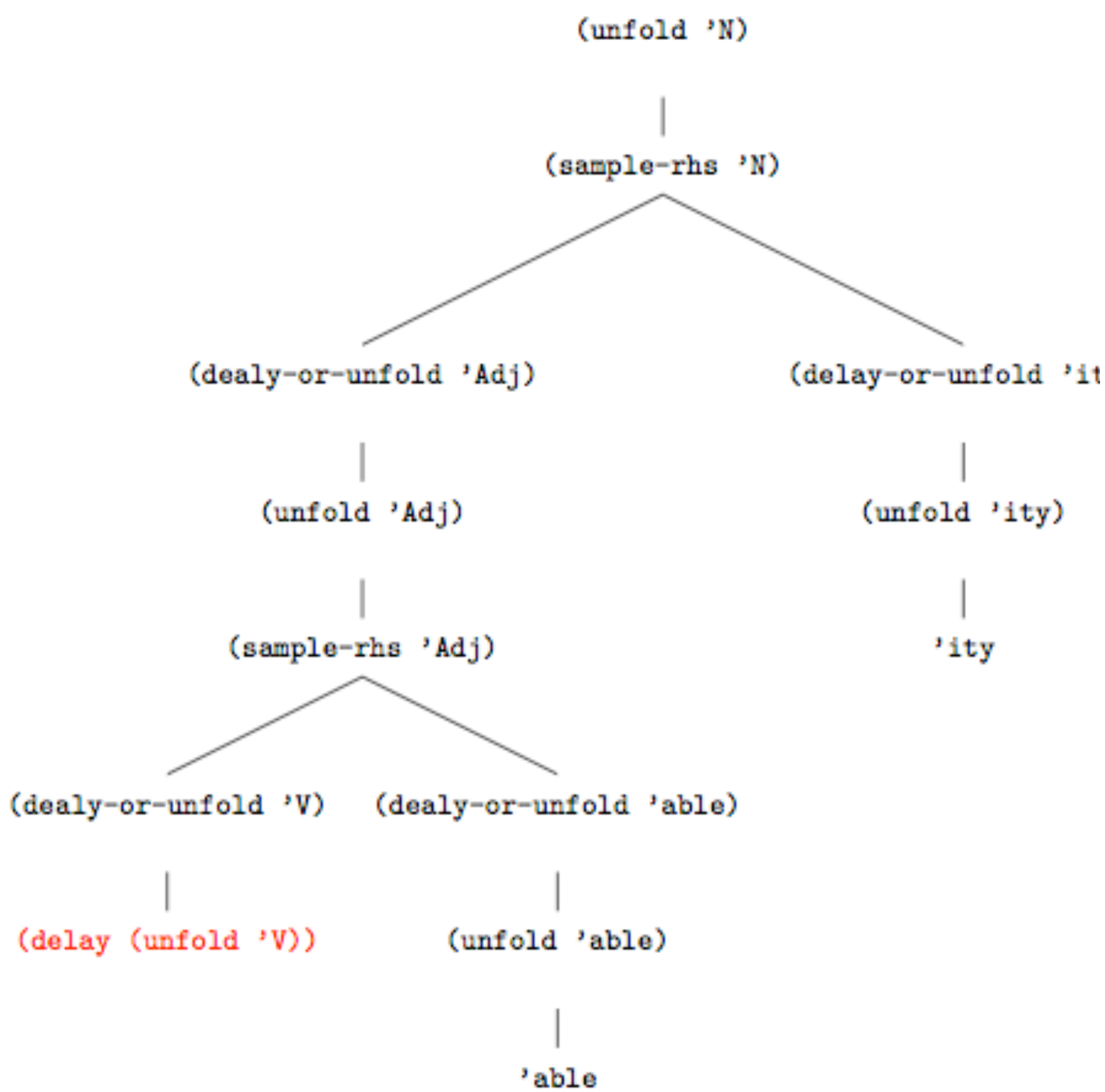
Stochastic Lazy `unfold`

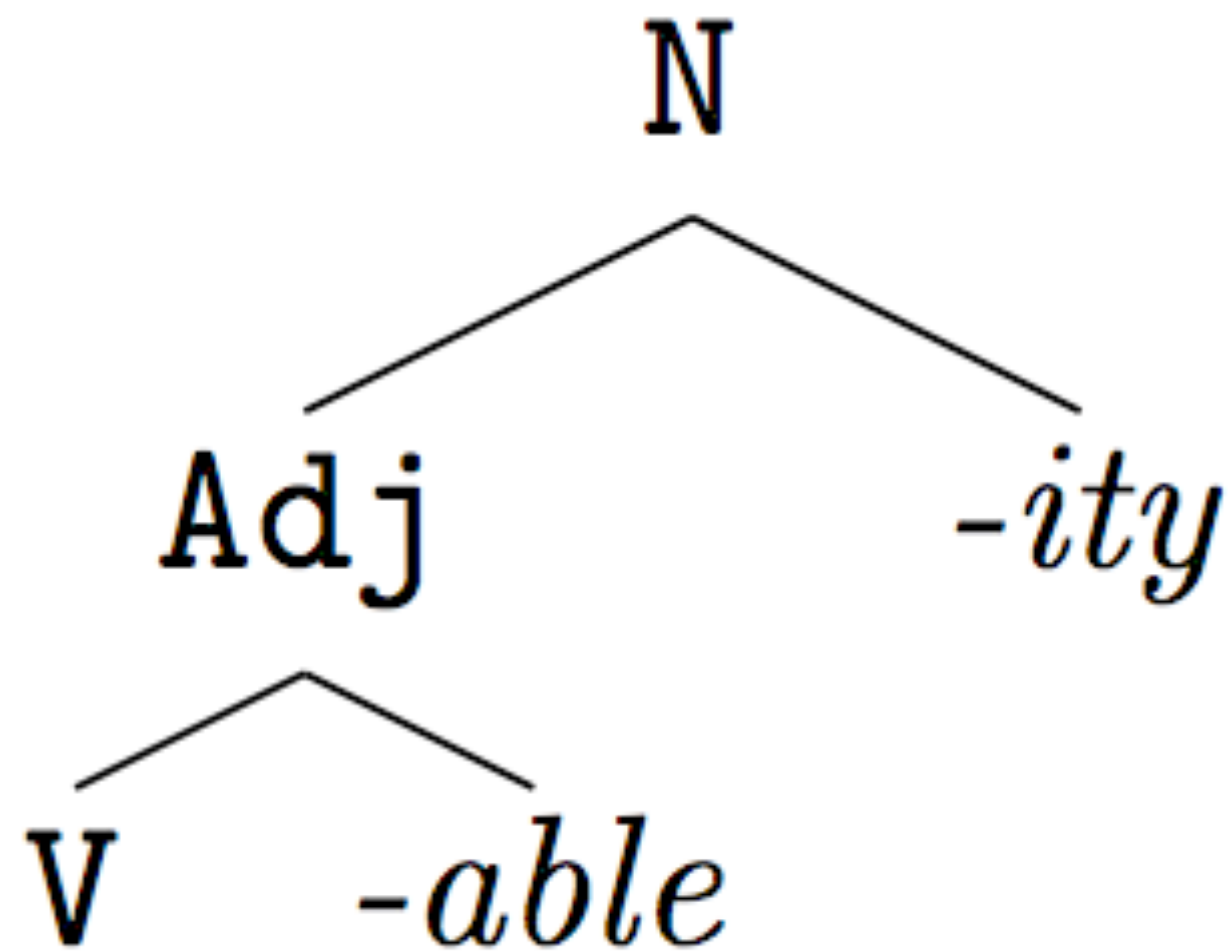
```
(define stochastic-lazy-unfold
  (lambda (symbol)
    (if (terminal? symbol)
      symbol
      (map delay-or-unfold (sample-rhs symbol)))))

(define delay-or-unfold
  (lambda (symbol)
    (if (flip)
      (delay (stochastic-lazy-unfold symbol))
      (stochastic-lazy-unfold symbol))))
```

Computation Trace with Delay



Computation Trace with Delay





Reusing Delayed Computations

Reusing Delayed Computations

- Need to be able to reuse partial evaluations.

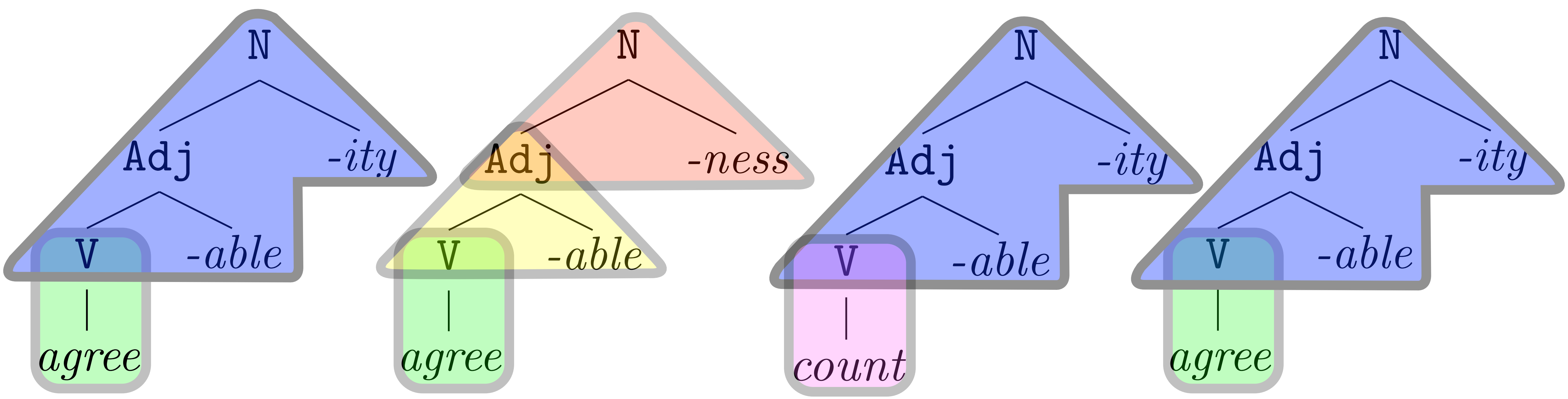
Reusing Delayed Computations

- Need to be able to reuse partial evaluations.
- Memoize stochastically lazy unfold.

Fragment Grammars

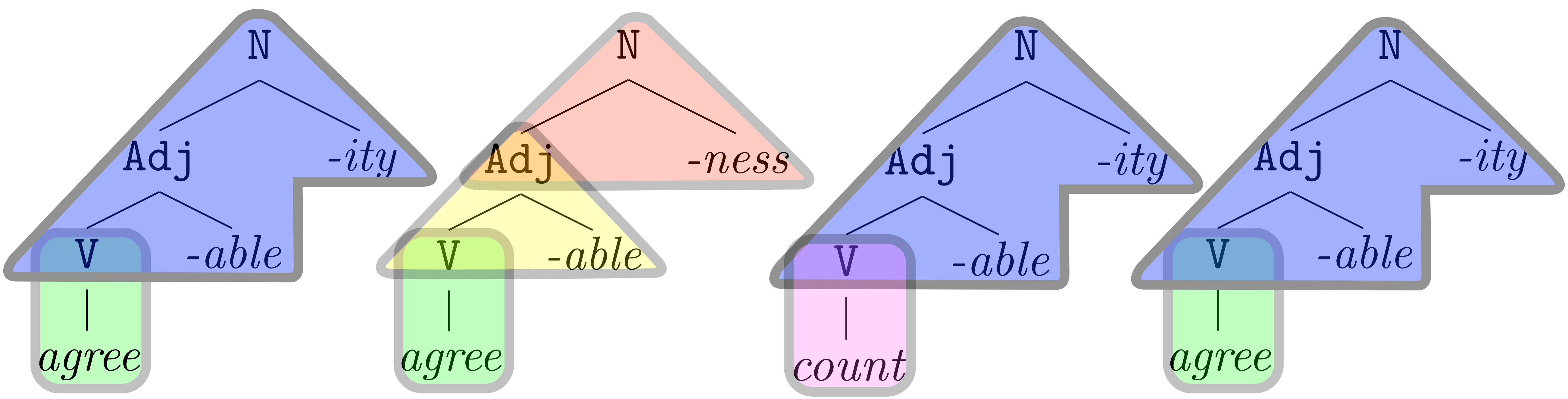
```
(define stochastic-lazy-unfold
  (lambda (symbol)
    (if (terminal? symbol)
      symbol
      (map delay-or-unfold (sample-rhs symbol)))))

(define delay-or-unfold
  (PYMem a b (lambda (symbol)
    (if (flip)
      (delay (stochastic-lazy-unfold symbol))
      (stochastic-lazy-unfold symbol)))))
```

Fragment Grammar
Reusable Computations



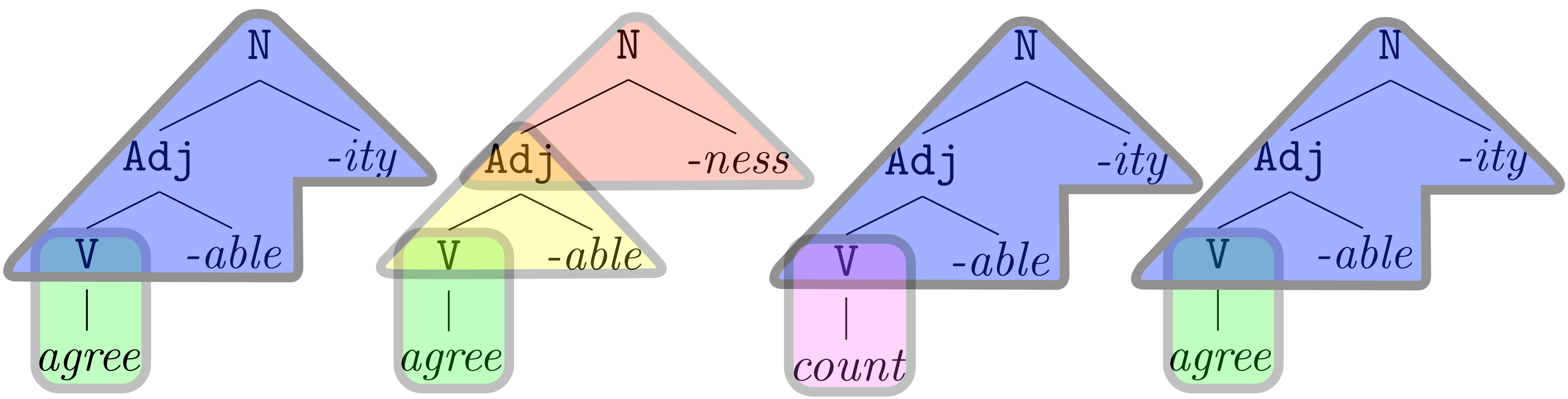
Fragment Grammar Reusable Computations

1. Always possible to use base grammar.



Fragment Grammar
Reusable Computations

1. Always possible to use base grammar.
2. Fully recursive.



Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. English Derivational Morphology
4. Conclusion

Five Models

Five Models

- 4 approaches to productivity and reuse.

Five Models

- 4 approaches to productivity and reuse.
- Capture historical proposals from the literature.

Five Models

- 4 approaches to productivity and reuse.
- Capture historical proposals from the literature.
- State-of-the-art probabilistic models.

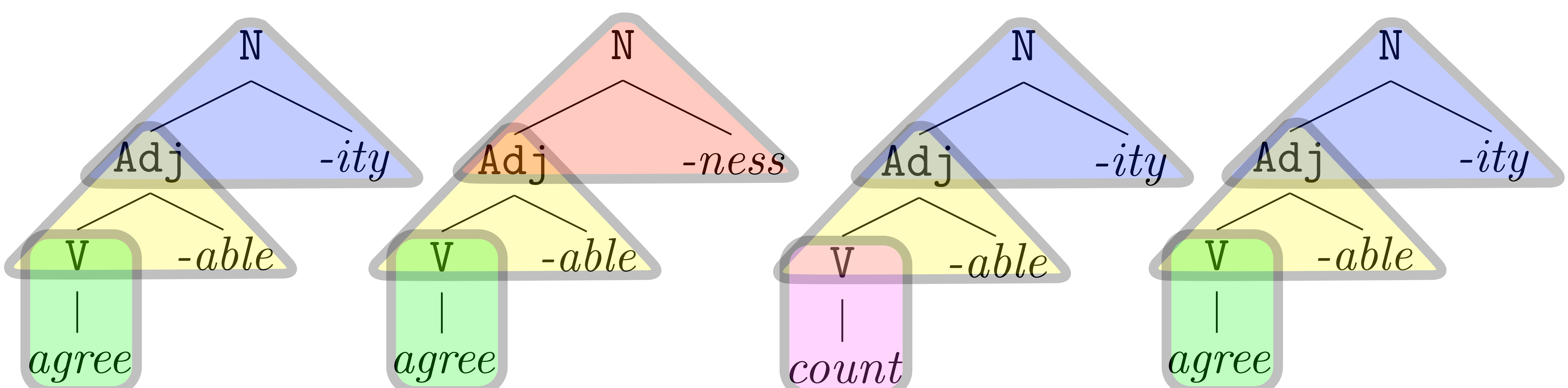
Five Models

- 4 approaches to productivity and reuse.
- Capture historical proposals from the literature.
- State-of-the-art probabilistic models.
    - Allow for variability and learning.

MDPCFG

Multinomial-Dirichlet Context-Free Grammars
(Full-Parsing)

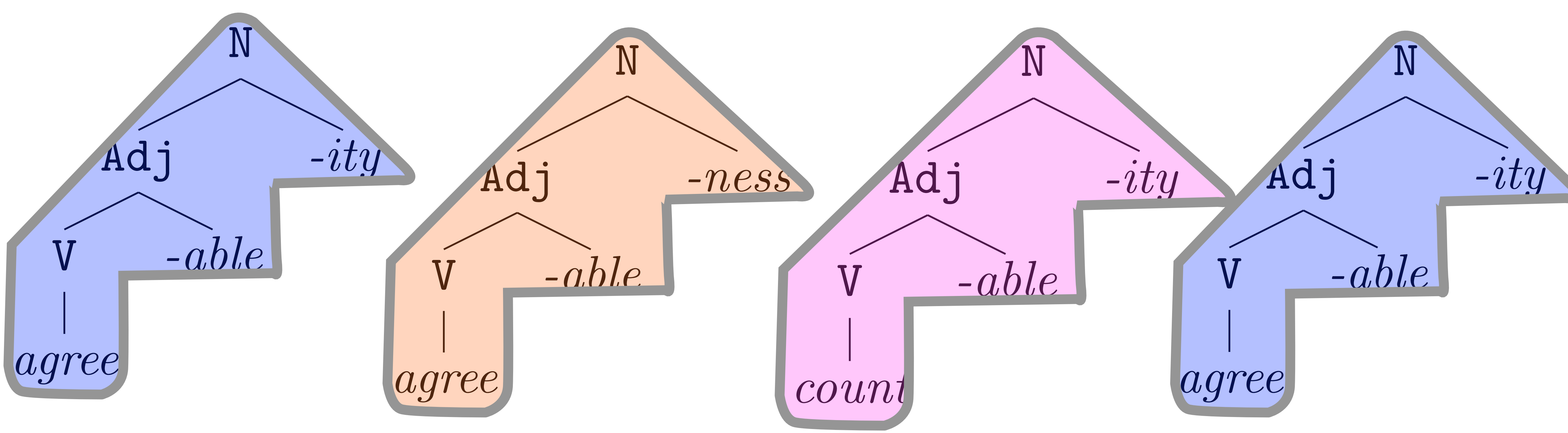
- All generalizations are productive
- Formalization: *Multinomial-Dirichlet Probabilistic Context-free Grammar* (MDPCFG; Johnson, et al. 2007a)



MAG

MAP Adaptor Grammars
(Full-entry)

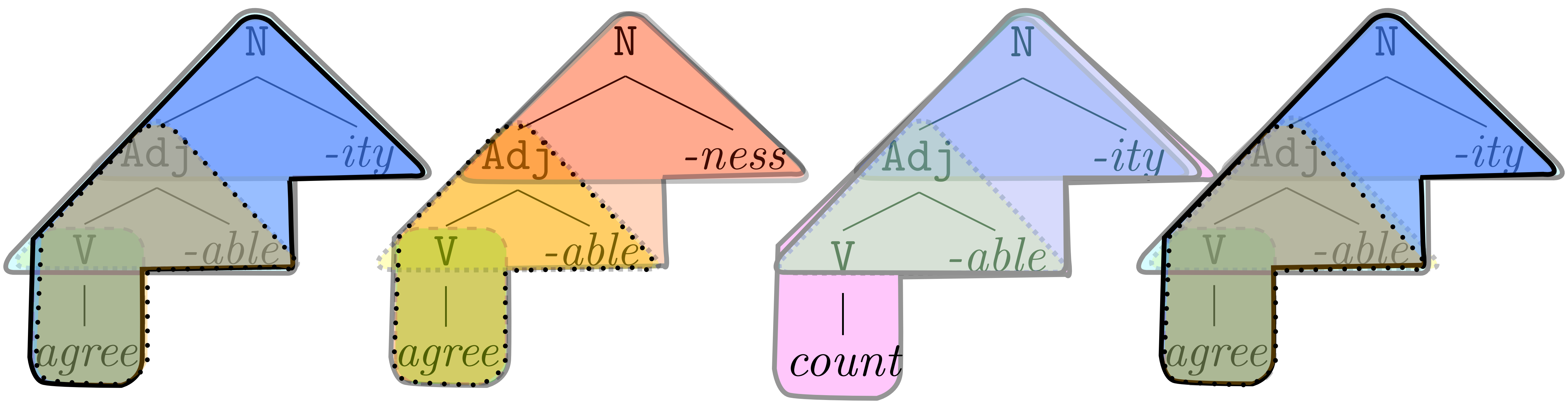
- Store whole form after *first* use.
- Formalization: *Adaptor Grammars* (AG; Johnson, et al. 2007).
- Always possible to compute productively with small probability; Fully recursive.
- Formalizes classic lexicalist theories (e.g., Jackendoff, 1975).



DOP1/GDMN

Data-Oriented Parsing
(Exemplar-based)

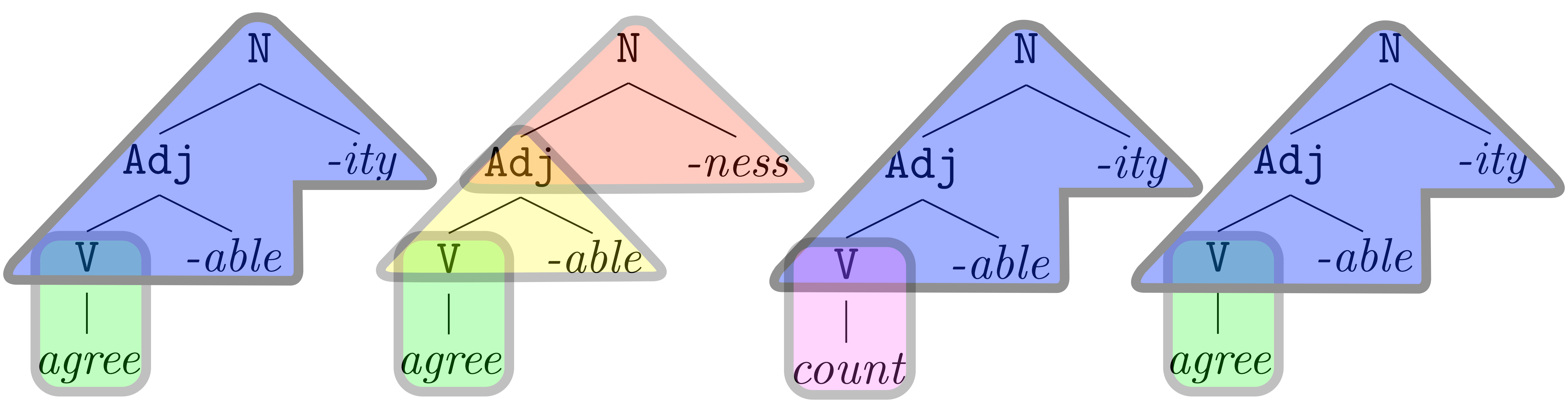
- Store *all* generalizations consistent with input
- Formalization: *Data-Oriented Parsing 1* (DOP1; Bod, 1998), *Data-Oriented Parsing: Goodman Estimator* (GDMN; Goodman, 2003)
- Recently proposed as models of syntax (e.g., Snider, 2009; Bod, 2009)



FG

Fragment Grammars
(Inference-based)

- Store *best* set of subcomputations for explaining the data.
- Formalization: *Fragment Grammars* (FG; O'Donnell, et al. 2009)
- Generalization of *Adaptor Grammars*



Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. English Derivational Morphology
4. Conclusion

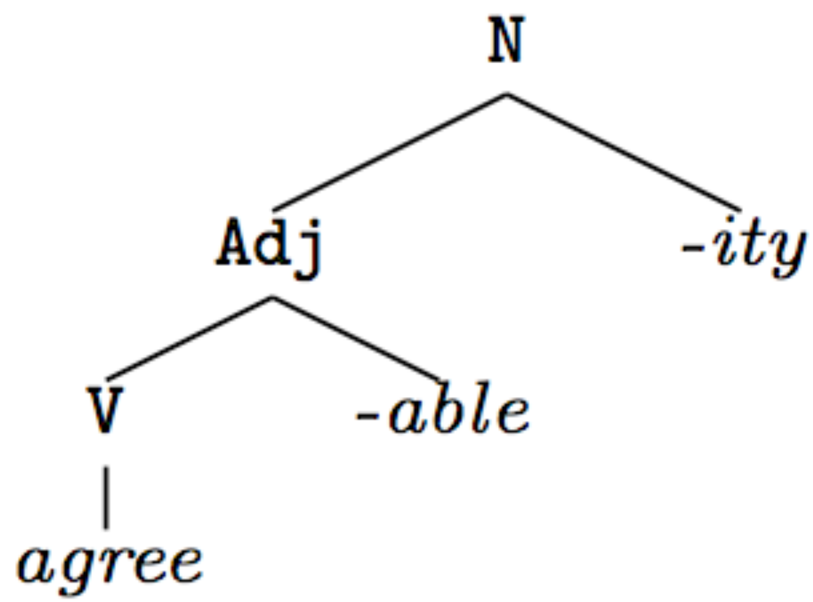
English Derivational Morphology

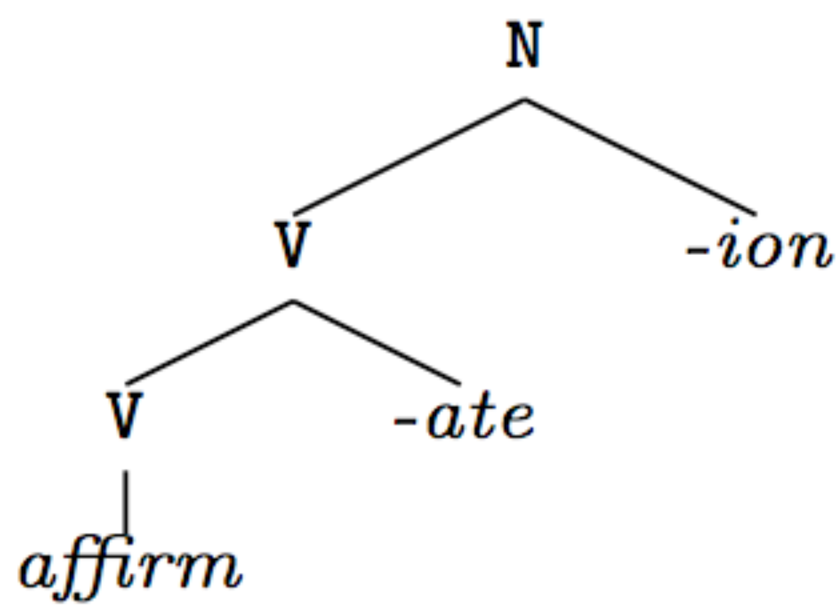
Productive	+ness *(goodness)*, +ly *(quickly)*
Semi-productive	+ity *(ability)*, +or *(operator)*
Unproductive	+th *(width)*

Simulations

- Words from CELEX.
- Extensive heuristic parsing/hand correction.
- Input format.
  - No phonology or semantics.

Derivational Inputs





English Derivational Morphology

Productive	+ness *(goodness)*, +ly *(quickly)*
Semi-productive	+ity *(ability)*, +or *(operator)*
Unproductive	+th *(width)*

1. Individual suffix productivity differences (-ness/-ity/-th).
2. Suffix sequences.

English Derivational Morphology

Productive	+ness *(goodness)*, +ly *(quickly)*
Semi-productive	+ity *(ability)*, +or *(operator)*
Unproductive	+th *(width)*

1. Individual suffix productivity differences (-ness/-ity/-th).
2. Suffix combinations.

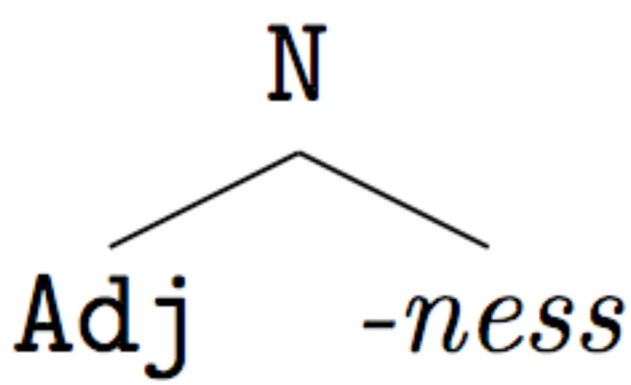
Productivity

- No gold-standard dataset or measure.
  - E.g., Large databases of *wug*-tests or naturalness judgments.
- Analyses.

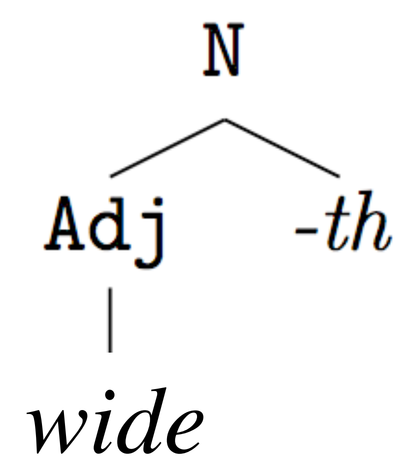
1. Convergence with other theoretical measures.

How is Productivity Represented?

- Relative probability of fragments with or without variables.



V.



Baayen's Corpus-Based Measures

- Baayen's $\mathcal{P}$ / $\mathcal{P}^*$ (e.g., Baayen, 1992)
  - $\mathcal{P}$: Prob(NOVEL | SUFFIX) i.e. rate of growth of forms with suffix
  - $\mathcal{P}^*$: Prob(SUFFIX | NOVEL) i.e. rate of growth of vocabulary due to suffix

Productivity Correlations

($\mathcal{P}/\mathcal{P}^*$ values from Hay & Baayen, 2002)

Model	FG (Inference-based)	MDPCFG (Full-parsing)	MAG (Full-listing)	DOP1 (Exemplar-based)	GDMN (Exemplar-based)
$\mathcal{P}$	**0.907**	-0.0003	0.692	0.346	0.143
$\mathcal{P}^*$	**0.662**	0.480	0.568	0.402	0.500

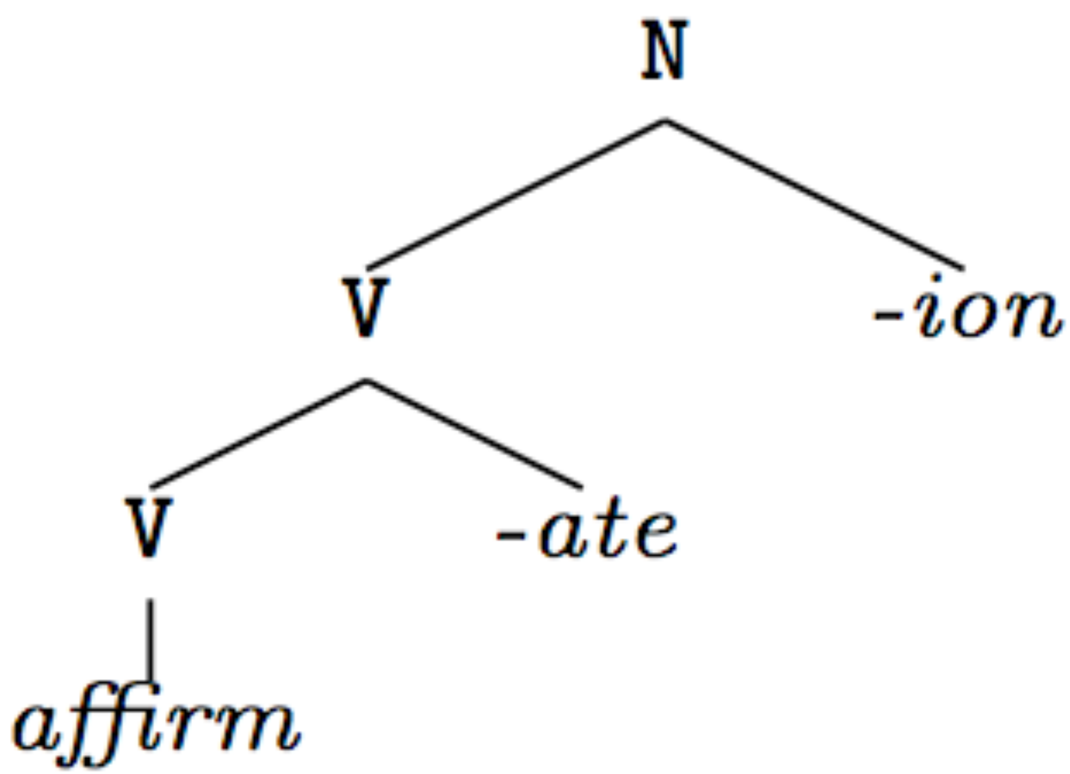
English Derivational Morphology

Productive	+ness *(goodness)*, +ly *(quickly)*
Semi-productive	+ity *(ability)*, +or *(operator)*
Unproductive	+th *(width)*

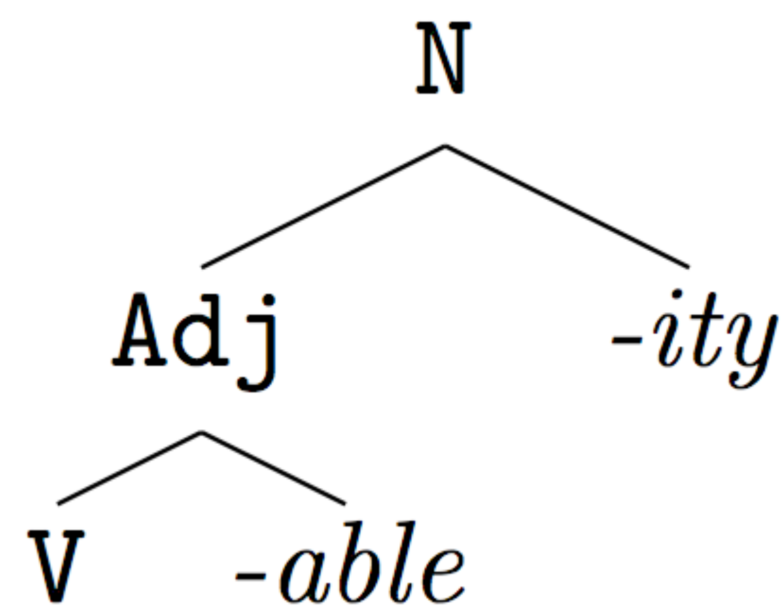
1. Individual suffix productivity differences (-ness/-ity/-th).
2. Suffix combinations.

Generalizable Combinations

Frozen Combinations

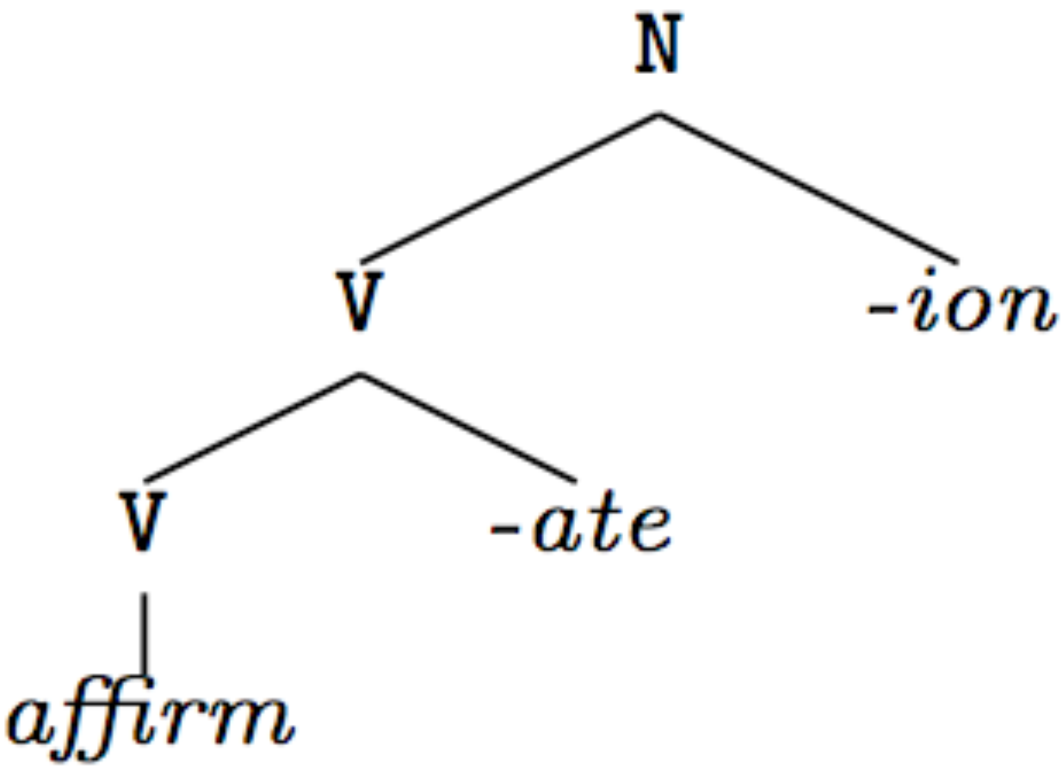


Generalizable Combinations

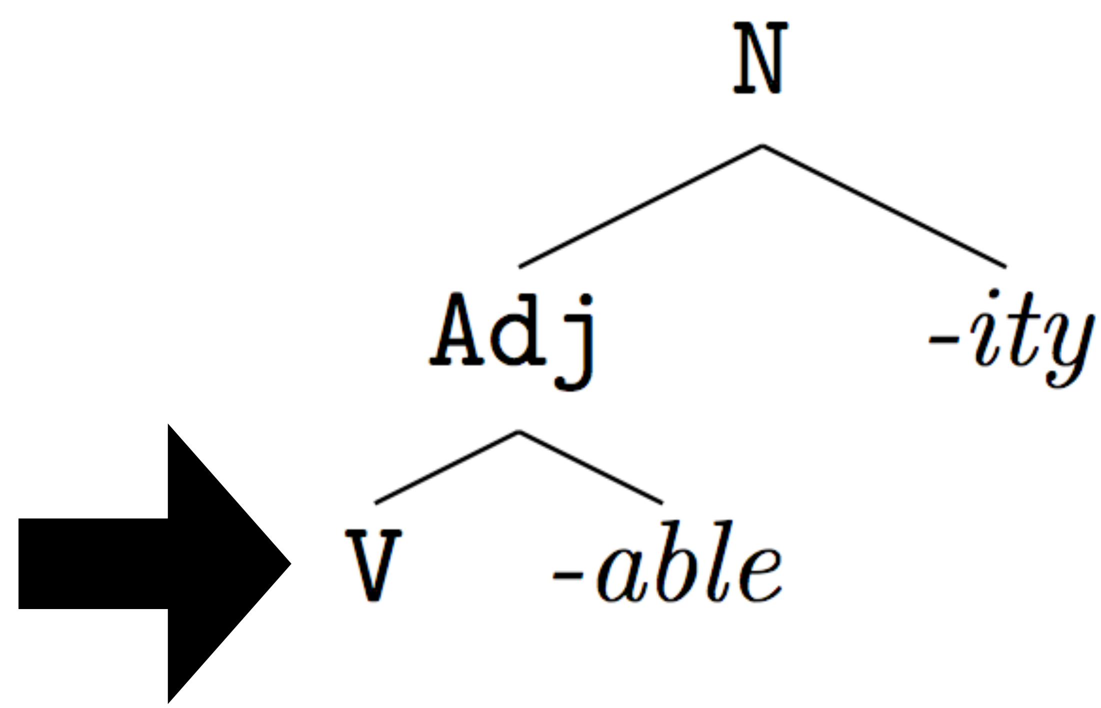


Generalizable Combinations

Frozen Combinations



Generalizable Combinations



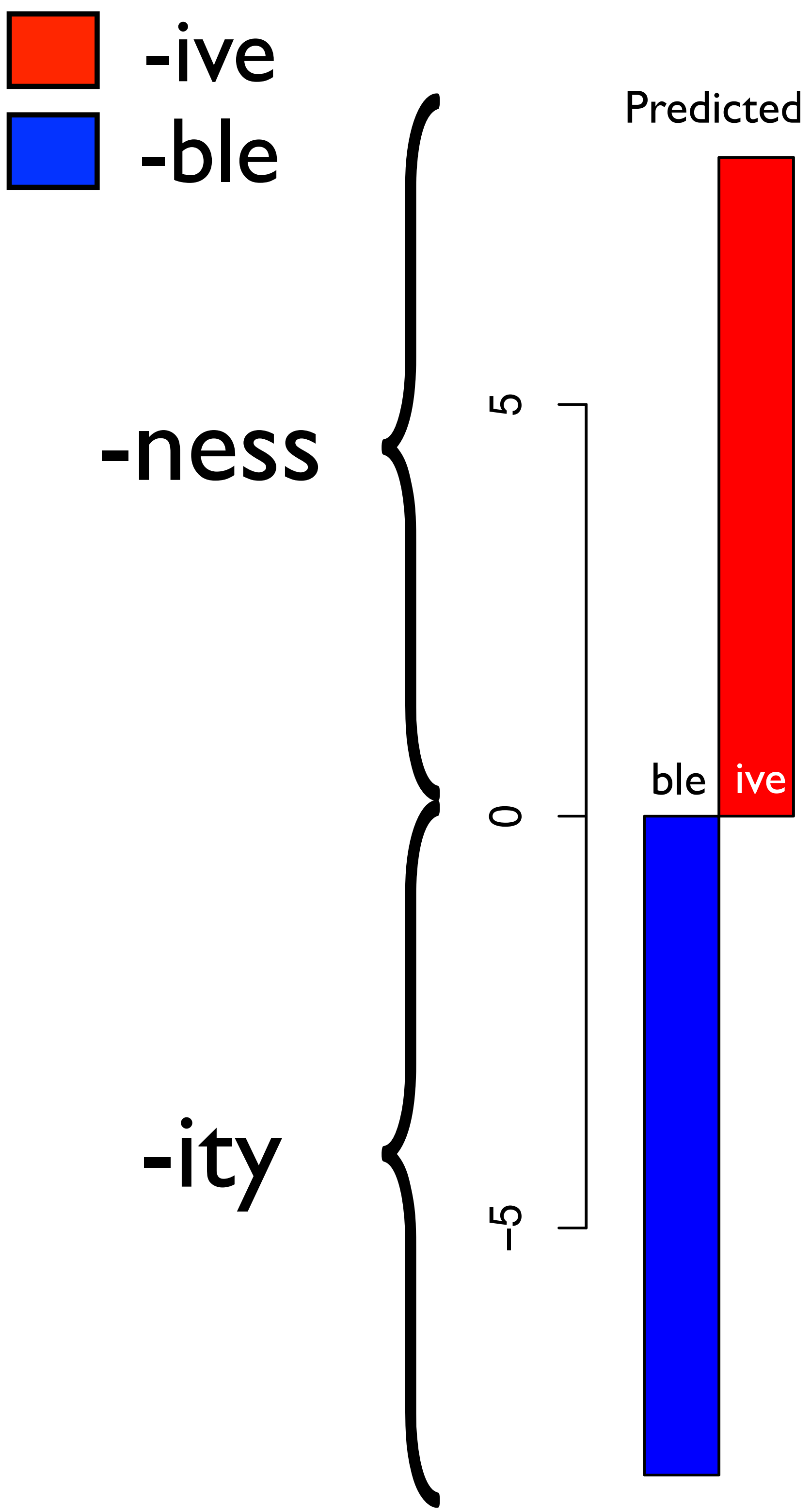
-ity v. -ness

- -ness more productive than -ity.
- -ity more productive than -ness after: -ile, -able, -(i)an, -ic.
  (Anshen & Aronoff, 1981; Aronoff & Schvaneveldt, 1978; Cutler, 1980)

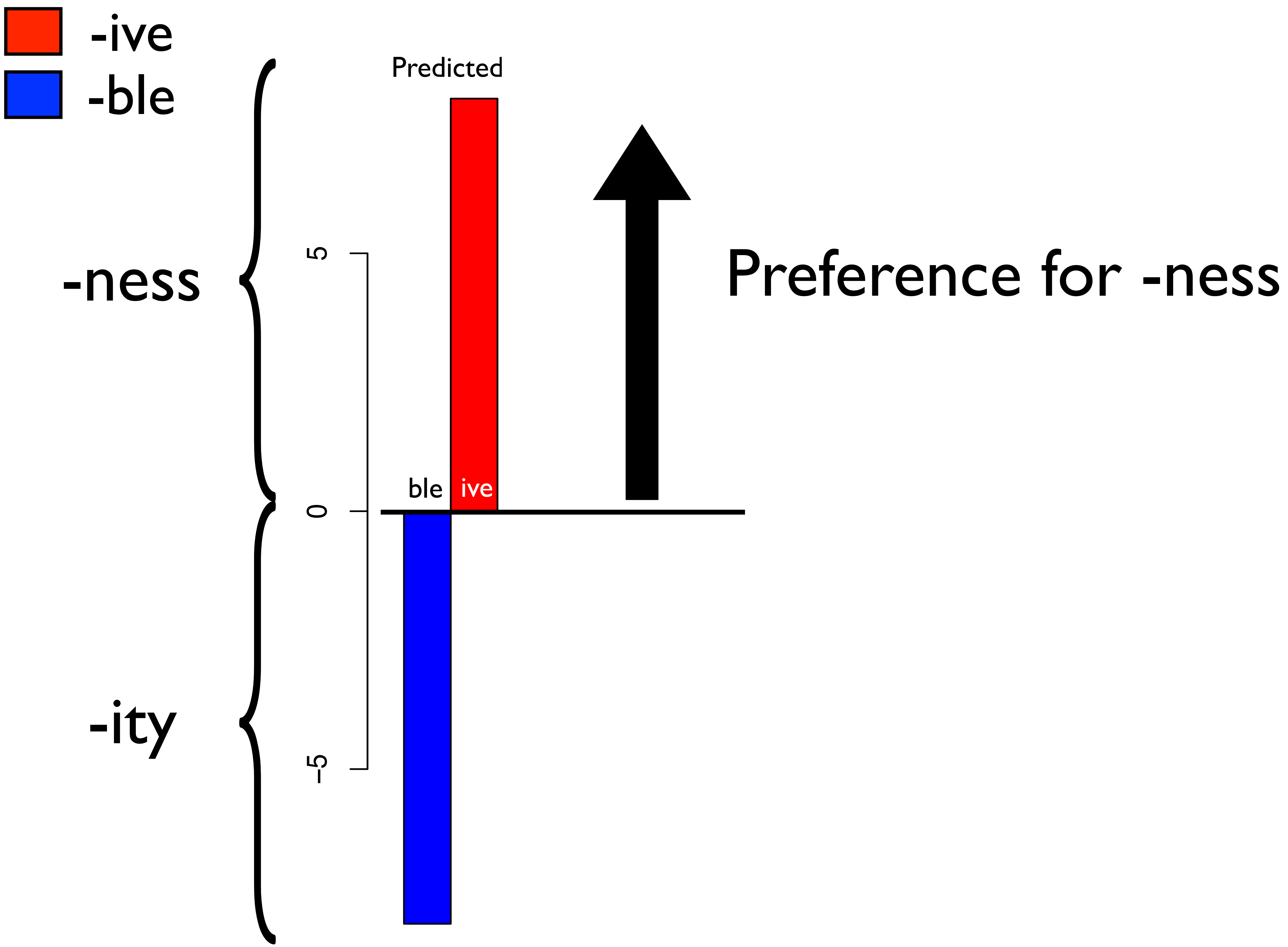
Two Frequent Combinations: -ivity v. -bility

- -ive + -ity: ***-ivity*** (e.g., selectivity).
  - Speaker prefer to use -ness with novel words (Aronoff & Schvaneveldt, 1978).
  - depulsiveness > depulsivity.
- -ble + -ity: ***-bility*** (e.g., sensibility).
  - Speakers prefer to use -ity with novel words (Anshen & Aronoff, 1981).
  - remortibility > remortibleness.

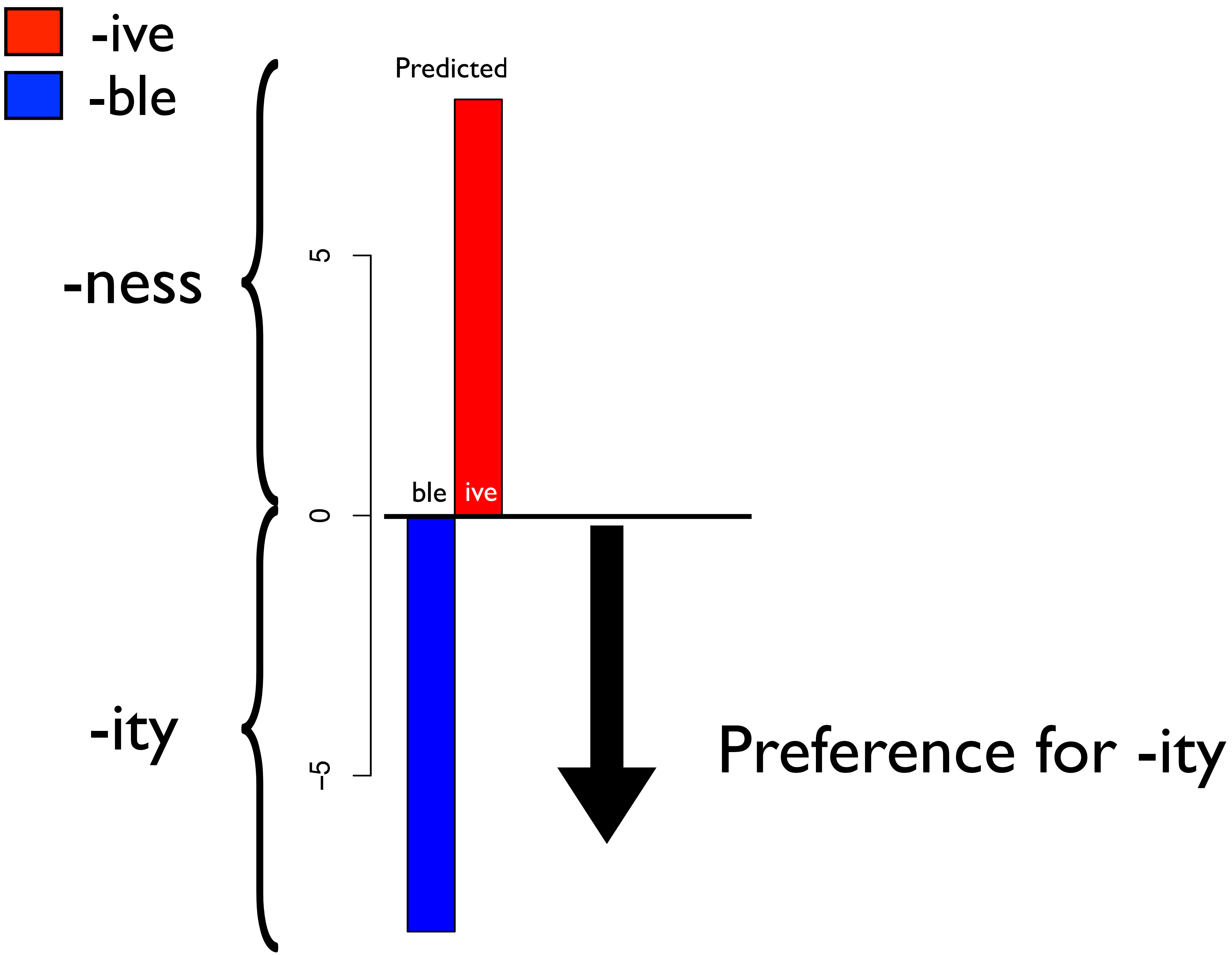
-ivity v. -bility



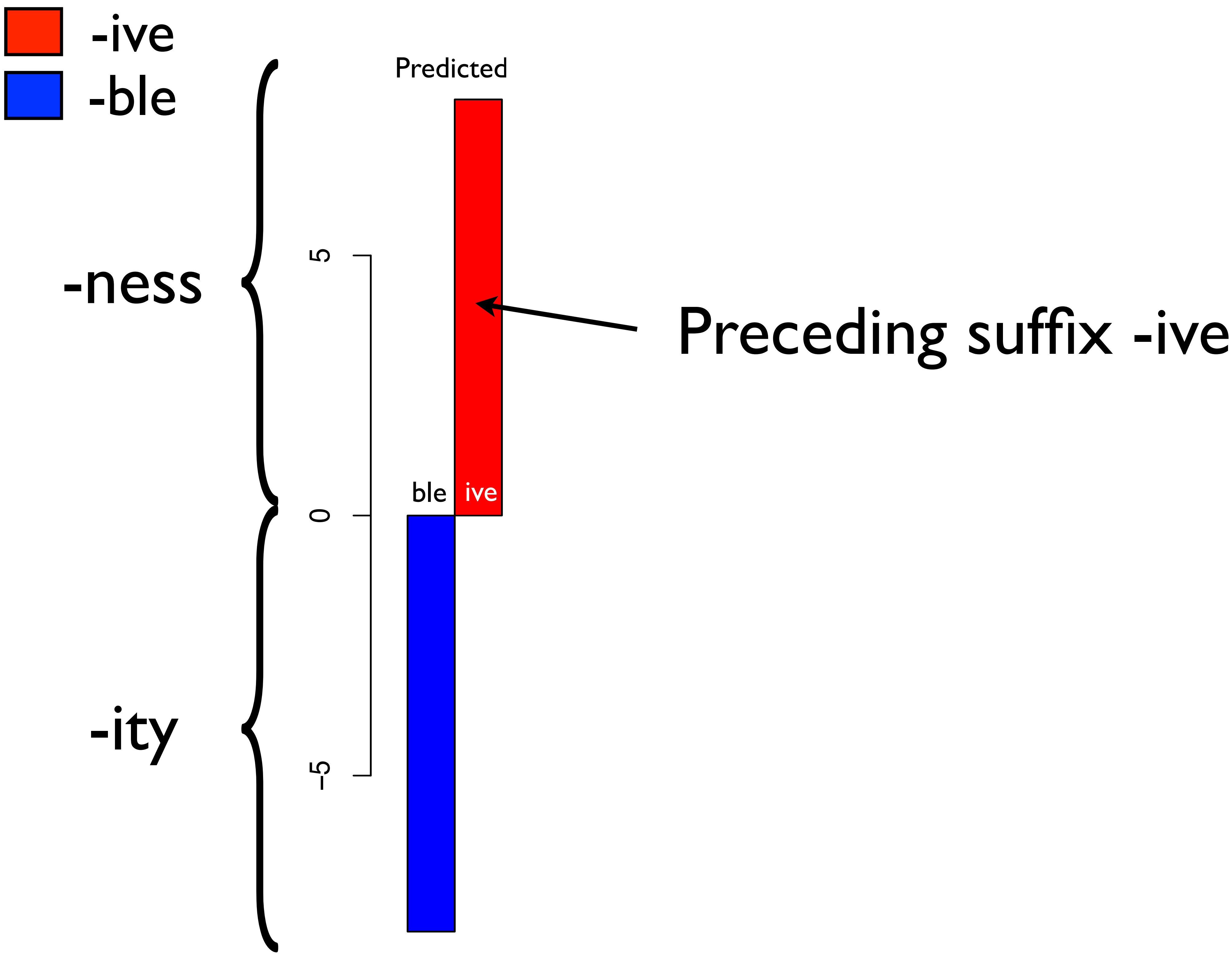
-ivity v. -bility



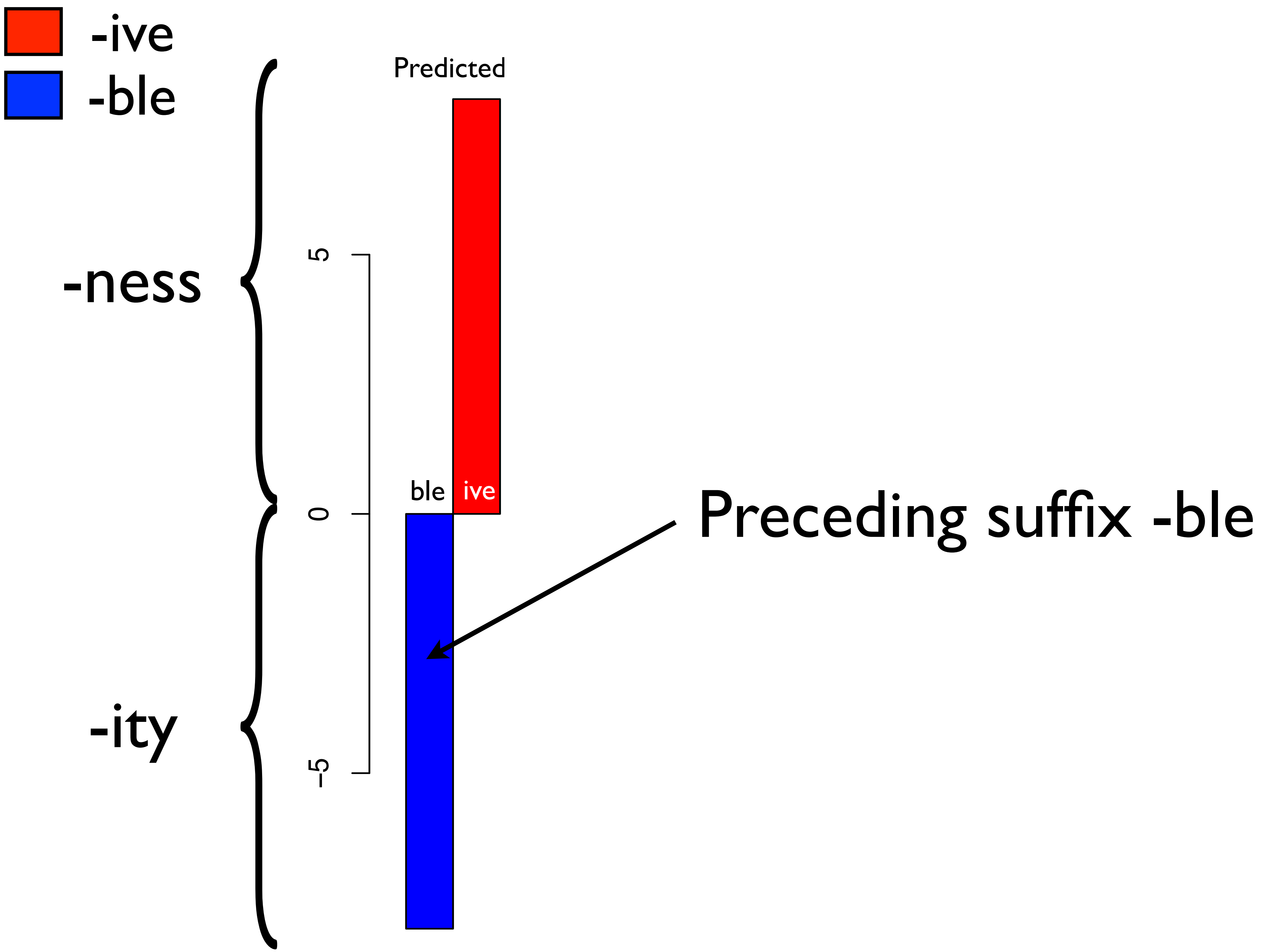
-ivity v. -bility



-ivity v. -bility

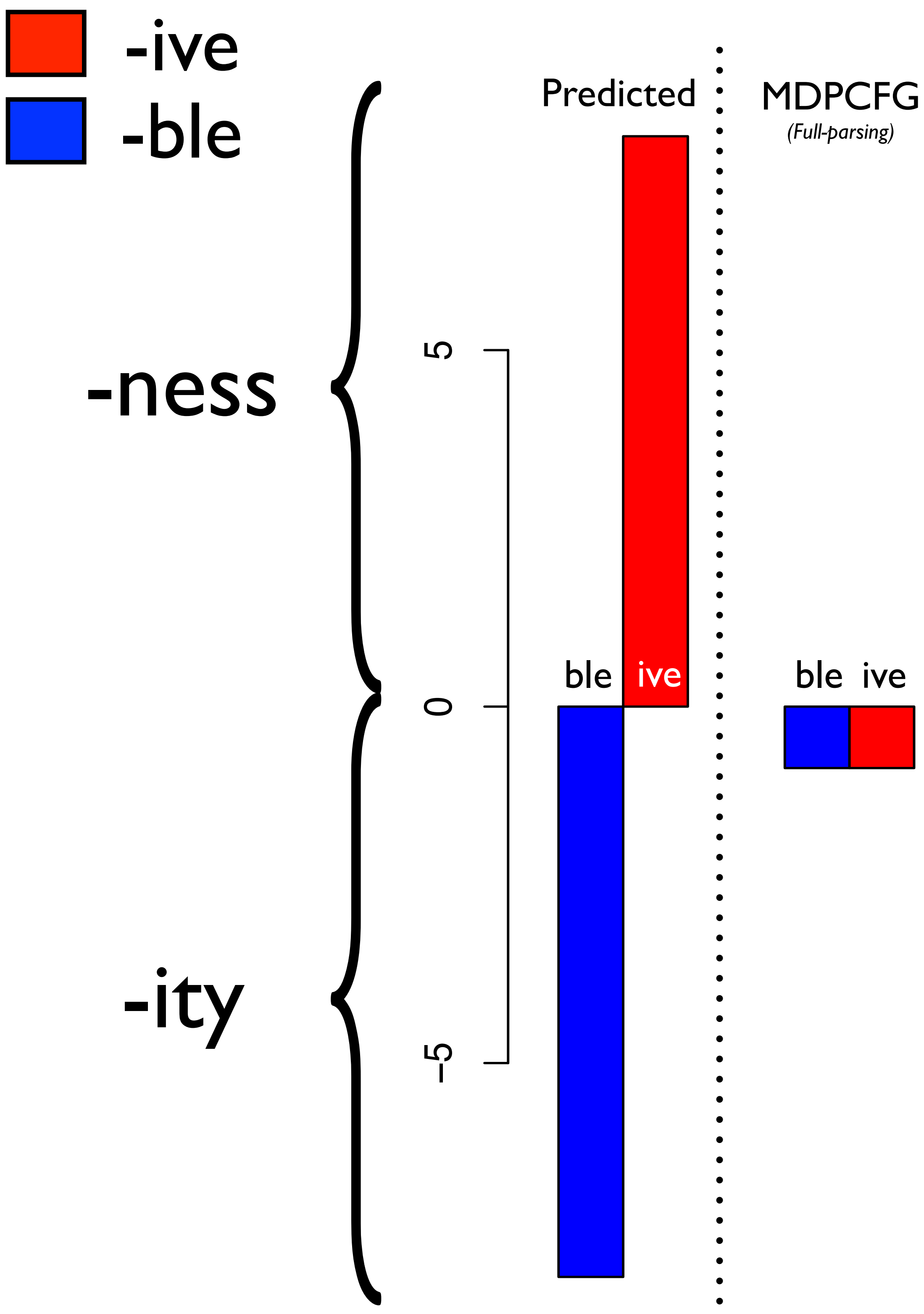


-ivity v. -bility



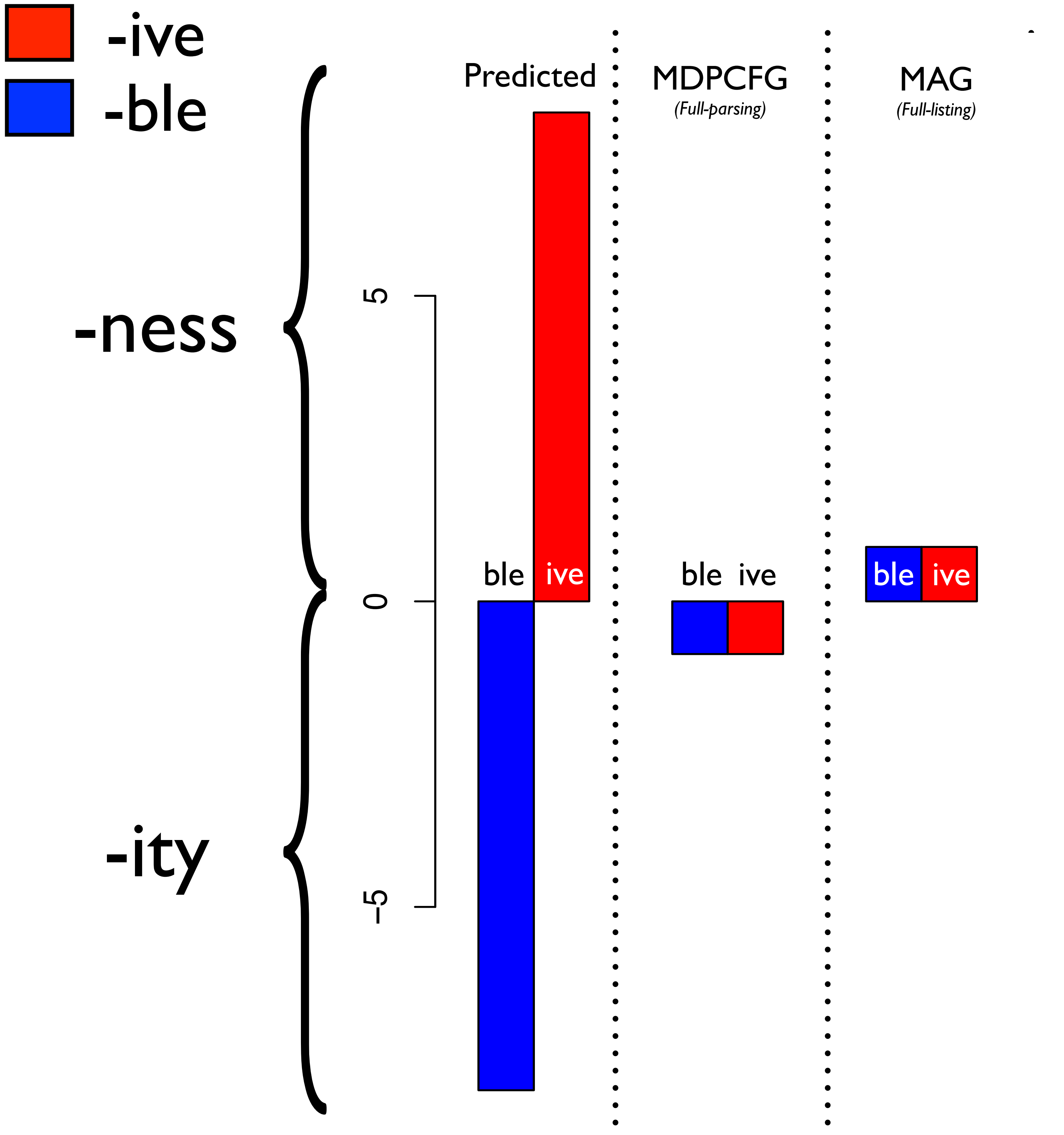
MDPCFG

(Full-parsing)



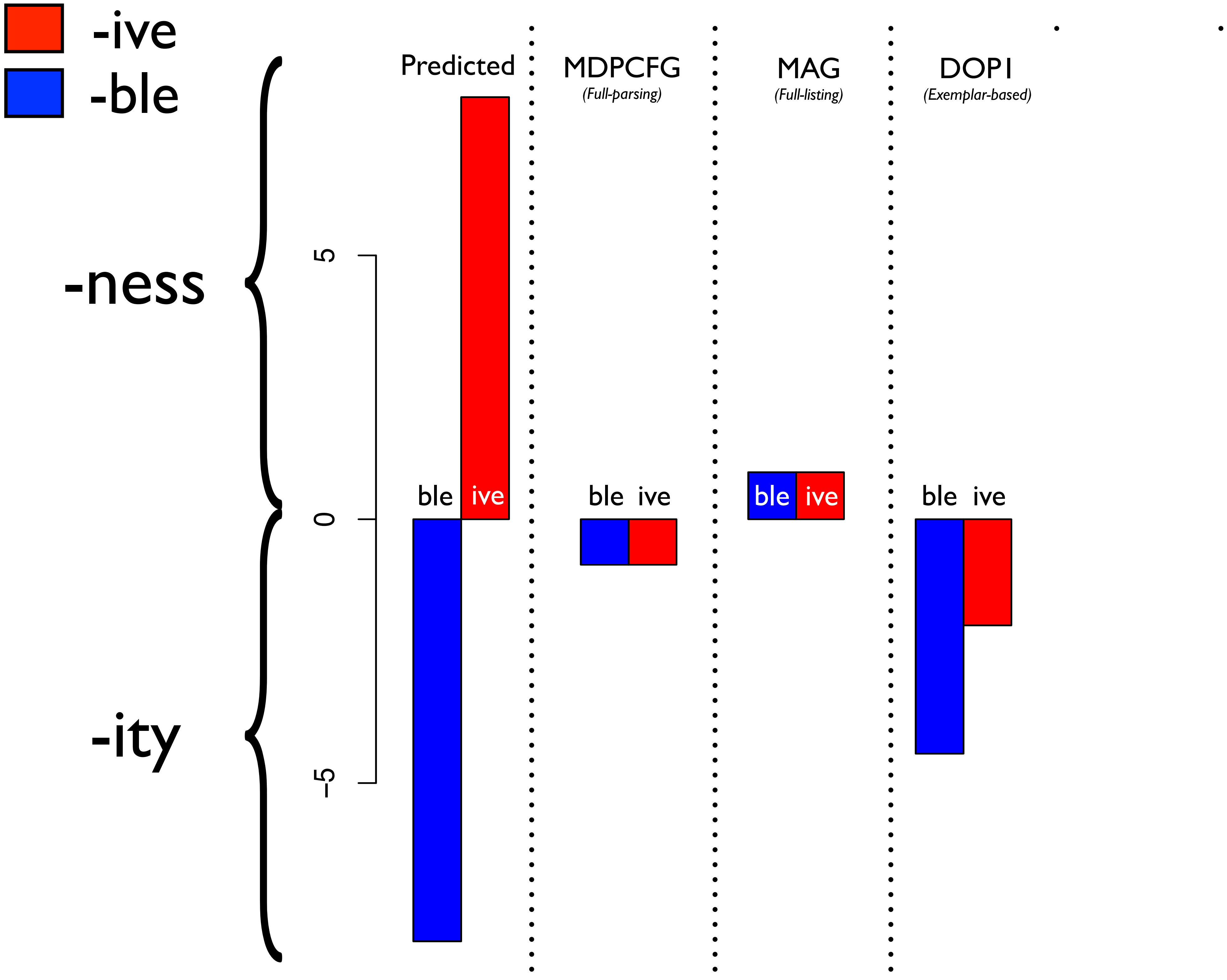
MAG

(Full-listing)



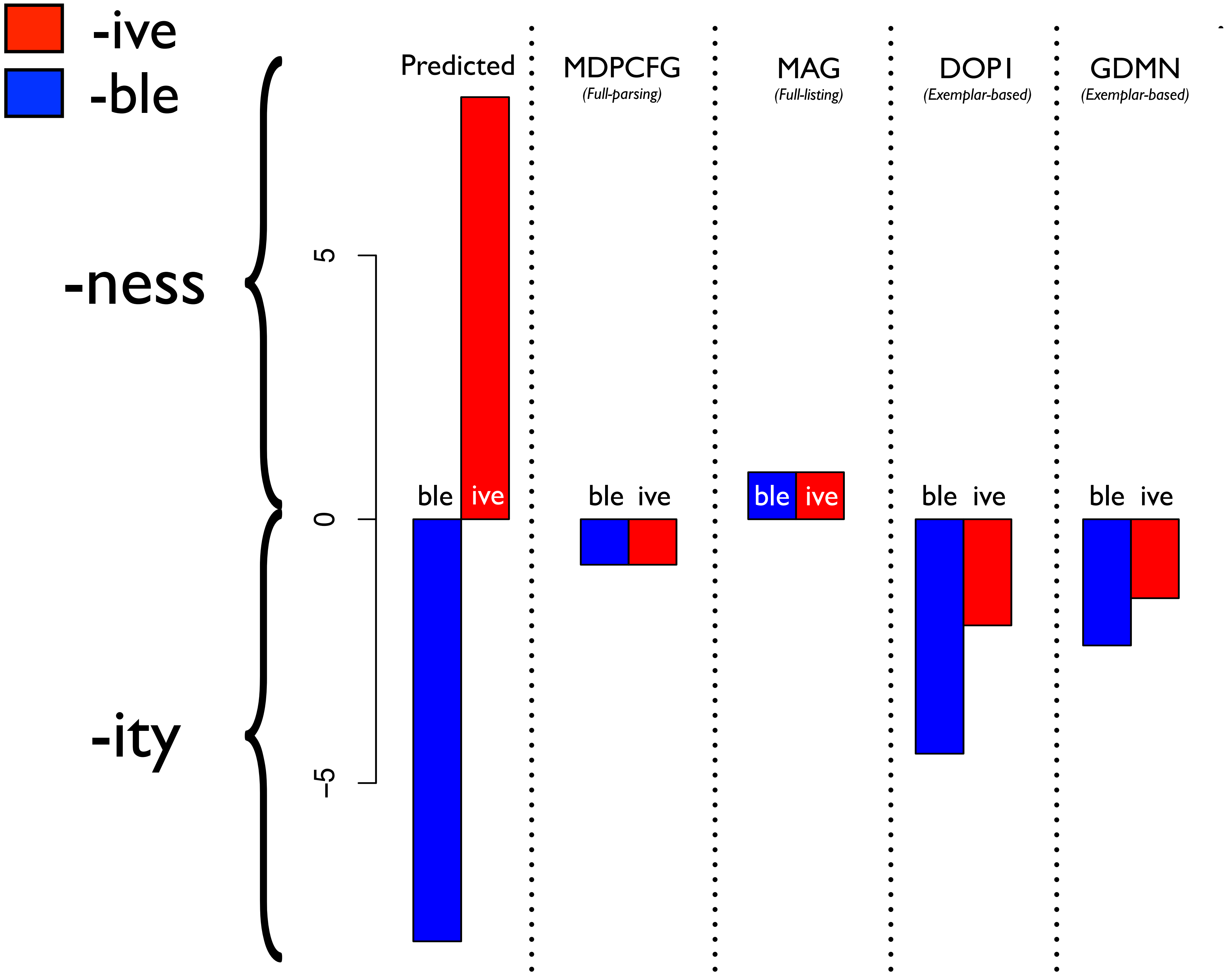
DOP1

(Exemplar-based)



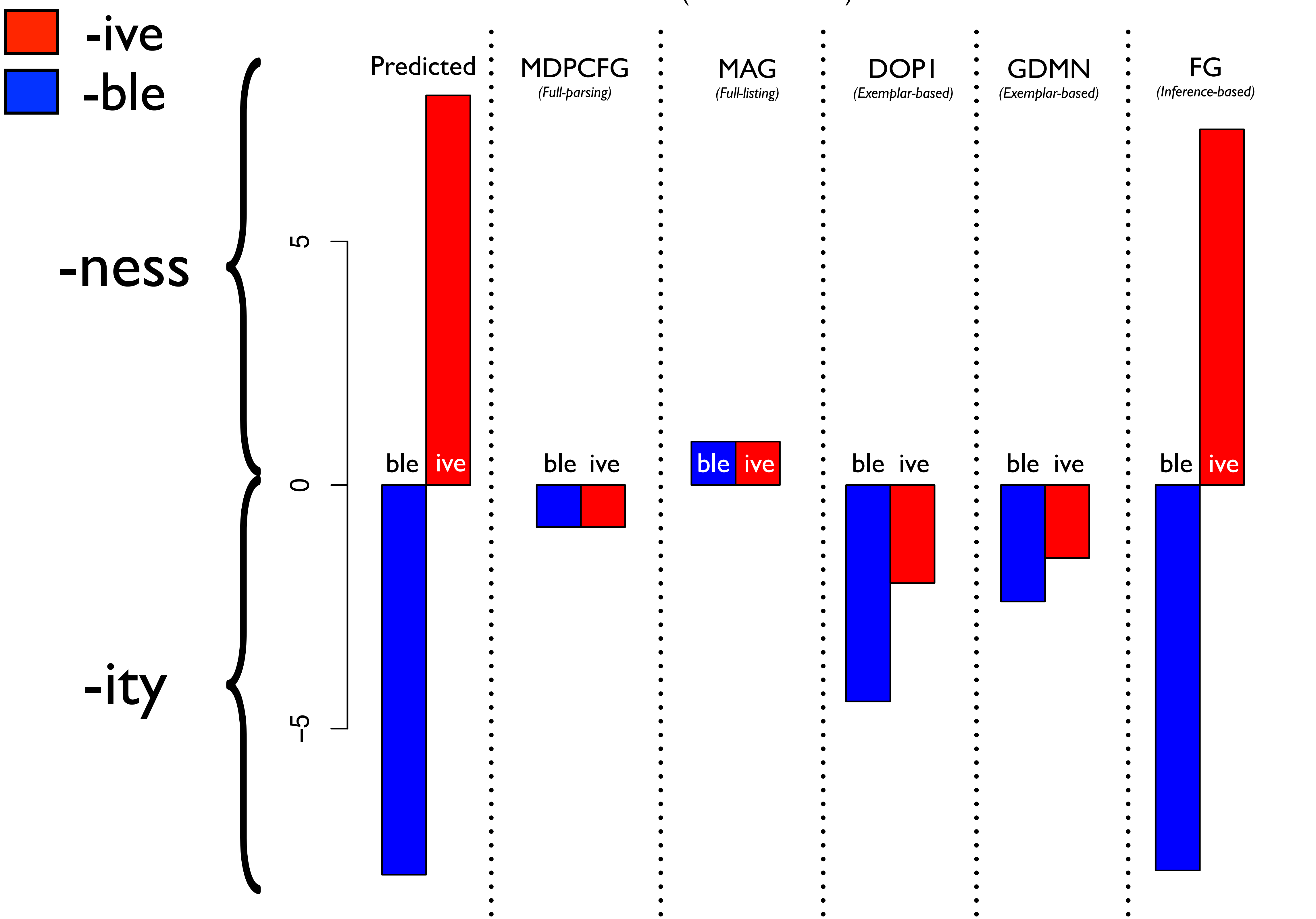
GDMN

(Exemplar-based)



FG

(Inference-based)



Discussion

- Inference-based approach able to correctly ignore high token frequency of -ivity because it balances a **tradeoff**.
- Other models use type or token frequencies.

Outline

1. The Proposal.
2. Five Models of Productivity and Reuse.
3. Empirical Evaluation

   The English Past Tense

   English Derivational Morphology
4. Conclusion

Conclusion

Conclusion

- View productivity and reuse as an inference.

Conclusion

- View productivity and reuse as an inference.
- Link between theory of programming languages and Bayesian models.

Conclusion

- View productivity and reuse as an inference.
- Link between theory of programming languages and Bayesian models.
- Able to capture dominant patterns **without** semantic and phonological structure.

Conclusion

- View productivity and reuse as an inference.
- Link between theory of programming languages and Bayesian models.
- Able to capture dominant patterns **without** semantic and phonological structure.
  - Future work...

Thanks!