Fragment Grammars: Productivity and Reuse in Language

Timothy J. O’Donnell
-ness
-ness

• circuitousness, grandness, orderliness, pretentiousness, cheapness, coolness, warmness, ...
-ness

- circuitousness, grandness, orderliness, pretentiousness, cheapness, coolness, warmth, ...

- Adj>N
-ness

- circuitousness, grandness, orderliness, pretentiousness, cheapness, coolness, warmthness, ...

- Adj>N

- grand + -ness
-ness

- circuitousness, grandness, orderliness, pretentiousness, cheapness, coolness, warmthness, ...

- 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/-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

- 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 \textit{novel} forms (e.g., -ness), which can only be \textit{reused} in existing forms (e.g., -th)?
Problem of Productivity

• Which processes can be used to construct \textit{novel} forms (e.g., -ness), which can only be \textit{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

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 → 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
...
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 combination
Subcomputations as Predictions

Prediction of future novelty/variability

-able

-ity

agree
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.
Languages for probability

- Purposes of a language:
  - Makes writing down models easier.
  - Makes reasoning about models clearer.
  - Supports efficient inference.
  - Gives ideas about mental representation.
\[ \lambda \text{ calculus} \]
\( \lambda \) calculus

- **Notation:**
  - Function have parentheses on the wrong side: \((\sin x)\)
  - Operators always go at the beginning: \((+ x y)\)
\( \lambda \) calculus

- **Notation:**
  - Function have parentheses on the wrong side: \((\sin x)\)
  - Operators always go at the beginning: \((+ x y)\)

- \( \lambda \) makes functions, define binds values to symbols:

\[
(\text{define } \text{double} \ (\lambda (x) (+ x x)))
\]
λ calculus

• Notation:
  • Function have parentheses on the wrong side:
  • Operators always go at the beginning:

  (sin x)

  (+ x y)

• λ makes functions, define binds values to symbols:

  (define double
   (λ (x) (+ x x)))

  (double 3) => 6
\textbf{\textlambda \text{ calculus}}

- Notation:
  - Function have parentheses on the wrong side: $(\sin x)$
  - Operators always go at the beginning: $(+ x y)$

- $\lambda$ makes functions, define binds values to symbols:

  \begin{align*}
  & (\text{define } \textbf{double} \\
  & \quad (\lambda (x) (+ x x))) \equiv 6 \\
  & (\text{define } \textbf{repeat} \\
  & \quad (\lambda (f) (\lambda (x) (f (f x)))))
  \end{align*}
λ calculus

- Notation:
  - Function have parentheses on the wrong side:
    (sin x)
  - Operators always go at the beginning:
    (+ x y)

- λ makes functions, define binds values to symbols:

  (define double
    (λ (x) (+ x x)))

  (define repeat
    (λ (f) (λ (x) (f (f x))))))

  ((repeat double) 3) => 12

  (double 3) => 6
λ calculus

- Notation:
  - Function have parentheses on the wrong side: \((\sin x)\), \((+ x y)\)
  - Operators always go at the beginning:

- \(\lambda\) makes functions, define binds values to symbols:

  
  \[
  (\text{define } \text{double} \ (\lambda (x) (+ x x)))
  \]

  \[
  (\text{double } 3) \Rightarrow 6
  \]

  
  \[
  (\text{define } \text{repeat} \ (\lambda (f) (\lambda (x) (f (f x)))))
  \]

  \[
  ((\text{repeat } \text{double}) 3) \Rightarrow 12
  \]

  
  \[
  (\text{define } \text{2nd-derivative} \ (\text{repeat } \text{derivative}))
  \]
ψλ-calculus

• How can we use these ideas to describe probabilities?

• ψλ-calculus: a stochastic variant.
  
  • We introduce a random primitive \texttt{flip}, such that \((\texttt{flip})\) reduces to a random sample \(t/f\).
  
  • The usual evaluation rules now result in \textit{sampled} values. This induces \textit{distributions}.

• This calculus, plus primitive operators and data types, gives the probabilistic programming language Church.
Church

Random primitives:

```
(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)
```
Random primitives:

\[
\begin{align*}
& (\text{define } \text{a} \ (\text{flip } 0.3)) \Rightarrow 1 \\
& (\text{define } \text{b} \ (\text{flip } 0.3)) \\
& (\text{define } \text{c} \ (\text{flip } 0.3)) \\
& (+ \text{a b c})
\end{align*}
\]
Church

Random primitives:

\[
\begin{align*}
\text{(define } & \mathbf{a} \ (\text{flip} \ 0.3)) \Rightarrow 1 \\
\text{(define } & \mathbf{b} \ (\text{flip} \ 0.3)) \Rightarrow 0 \\
\text{(define } & \mathbf{c} \ (\text{flip} \ 0.3)) \\
(+ & \ a \ b \ c)
\end{align*}
\]

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Church

Random primitives:

(define a (flip 0.3)) => 1
(define b (flip 0.3)) => 0
(define c (flip 0.3)) => 1
(+ a b c)
Church

Random primitives:

```
(define a (flip 0.3)) => 1
(define b (flip 0.3)) => 0
(define c (flip 0.3)) => 1
(+ a b c) => 2
```
Church

Random primitives:

- (define a (flip 0.3)) => 1 0
- (define b (flip 0.3)) => 0 0
- (define c (flip 0.3)) => 1 0
- (+ a b c) => 2 0

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Random primitives:

\[
\begin{align*}
\text{(define } a \ (\text{flip} \ 0.3)) & \Rightarrow 1 \ 0 \ 0 \\
\text{(define } b \ (\text{flip} \ 0.3)) & \Rightarrow 0 \ 0 \ 0 \\
\text{(define } c \ (\text{flip} \ 0.3)) & \Rightarrow 1 \ 0 \ 1 \\
(+ \ a \ b \ c) & \Rightarrow 2 \ 0 \ 1
\end{align*}
\]
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c) => 2 0 1..
Random primitives:

\[
\begin{align*}
\text{(define } a \text{ (flip 0.3))} &\Rightarrow 1 \ 0 \ 0 \\
\text{(define } b \text{ (flip 0.3))} &\Rightarrow 0 \ 0 \ 0 \\
\text{(define } c \text{ (flip 0.3))} &\Rightarrow 1 \ 0 \ 1 \\
(+ a \ b \ c) &\Rightarrow 2 \ 0 \ 1 \ldots
\end{align*}
\]
**Theorem**: Any computable distribution can be represented by a Church expression.

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Church

Random primitives:

\[
\begin{align*}
\text{(define } a & \text{ (flip 0.3))} & \Rightarrow 1 \ 0 \ 0 \\
\text{(define } b & \text{ (flip 0.3))} & \Rightarrow 0 \ 0 \ 0 \\
\text{(define } c & \text{ (flip 0.3))} & \Rightarrow 1 \ 0 \ 1 \\
\text{(+ a b c)} & \Rightarrow 2 \ 0 \ 1 \ldots
\end{align*}
\]

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Random primitives:

(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c) => 2 0 1..
Church

Random primitives:

\[(\text{define } a (\text{flip } 0.3)) \Rightarrow 1 0 0\]
\[(\text{define } b (\text{flip } 0.3)) \Rightarrow 0 0 0\]
\[(\text{define } c (\text{flip } 0.3)) \Rightarrow 1 0 1\]
\[(+ a b c) \Rightarrow 2 0 1\]

Conditioning (inference):

\[(\text{query} \quad (\text{define } a (\text{flip } 0.3)) \quad (\text{define } b (\text{flip } 0.3)) \quad (\text{define } c (\text{flip } 0.3)) \quad (+ a b c) \quad (= (+ a b) 1))\]

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Random primitives:

```
(define a (flip 0.3)) => 1 0 0
(define b (flip 0.3)) => 0 0 0
(define c (flip 0.3)) => 1 0 1
(+ a b c) => 2 0 1...
```

Conditioning (inference):

```
(query
  (define a (flip 0.3))
  (define b (flip 0.3))
  (define c (flip 0.3))
  (+ a b c) Query
  (= (+ a b) 1) Condition, must be true
)
```

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
Random primitives:

(define a (flip 0.3))
(define b (flip 0.3))
(define c (flip 0.3))
(+ a b c)

Conditioning (inference):

(query
 (define a (flip 0.3))
 (define b (flip 0.3))
 (define c (flip 0.3))
 (+ a b c))

Condition, must be true

Goodman, Mansinghka, Roy, Bonawitz, Tenenabum (2008)
• Universal inference: an algorithm that does inference for any Church query. (And hopefully is efficient for a wide class.)
• As a modeler, save implementation time: rapid prototyping.
• For cognitive science, shows that the mind could be a universal inference engine.
Example: Bayes Net
Example: Bayes Net

(define flu (flip 0.2))
(define TB (flip 0.01))
(define cough
  (if (or flu TB)
      (flip 0.8) (flip 0.1)))
“Infer the chance of flu, given observed cough.”

(define flu (flip 0.2))
(define TB (flip 0.01))
(define cough
  (if (or flu TB)
      (flip 0.8) (flip 0.1)))
“Infer the chance of flu, given observed cough.”

(query
  (define flu (flip 0.2))
  (define TB (flip 0.01))
  (define cough
    (if (or flu TB)
        (flip 0.8) (flip 0.1)))
  flu
  cough)
Example: Bayes Net

“Infer the chance of flu, given observed cough.”

```
(query
 (define flu (flip 0.2))
 (define TB  (flip 0.01))
 (define cough
   (if (or flu TB)
       (flip 0.8) (flip 0.1)))
 flu
cough)
```

=> true 66%
Example: Bayes Net

“Infer the chance of flu, given observed cough.”

(query
  (define flu (flip 0.2))
  (define TB (flip 0.01))
  (define cough
    (if (or flu TB)
        (flip 0.8) (flip 0.1)))
  flu
  (and cough TB))

=> true 66%
Example: Bayes Net

“\textit{Infer} the chance of flu, given observed cough.”

\begin{verbatim}
(query
  (define flu (flip 0.2))
  (define TB  (flip 0.01))
  (define cough
    (if (or flu TB)
        (flip 0.8) (flip 0.1)))
  flu
  (and cough TB))
\end{verbatim}

\begin{itemize}
  \item \textit{true} 20\%
\end{itemize}
"Infer the chance of flu, given observed cough."

```
(query
  (define flu (flip 0.2))
  (define TB (flip 0.01))
  (define cough
    (if (or flu TB)
      (flip 0.8) (flip 0.1)))
  flu
  (and cough TB))
 => true 20%
```
Example: Bayes Net

“Infer the chance of flu, given observed cough.”

(query
  (define flu (flip 0.2))
  (define TB (flip 0.01))
  (define cough
    (if (or flu TB)
      (flip 0.8) (flip 0.1)))
  flu
  (and cough TB)) => true 20%
“Infer the chance of flu, given observed cough.”

(query
  (define flu (flip 0.2))
  (define TB (flip 0.01))
  (define cough
    (if (or flu TB)
        (flip 0.9) (flip 0.1)))
  flu
  (and cough TB)) => true 20%
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.
Goals
Goals

1. Get across intuitions.
Goals

1. Get across intuitions.

2. Give flavor of relationships between modeling ideas and programming ideas.
(define unfold
  (lambda (symbol)
    (if (terminal? symbol)
        symbol
        (map unfold (sample-rhs symbol))))))
(define adapted-unfold
  (PYMem a b
    (lambda (symbol)
      (if (terminal? symbol)
          symbol
          (map unfold (sample-rhs symbol))))))
(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))))))
\[ G^a_{\text{pcfg}}(d) = \begin{cases} 
\sum_{r \in R_G : a \rightarrow \text{root}(\hat{d}_i), \ldots, \text{root}(\hat{d}_k)} \theta_r \prod_{i=1}^{k} G^\text{root}(\hat{d}_i)(\hat{d}_i) & \text{root}(d) = a \in V_G \\
1 & \text{root}(d) = a \in T_G 
\end{cases} \]
\[
G_{\text{AG}}^a(d) = \begin{cases} 
\sum_{r \in R_G : a \rightarrow \text{root}(\hat{d}_i), \ldots, \text{root}(\hat{d}_k)} \theta_r \prod_{i=1}^{k} \text{mem}\{G_{\text{AG}}^{\text{root}(\hat{d}_i)}\}(\hat{d}_i) & \text{root}(d) = a \in V_G \\
1 & \text{root}(d) = a \in T_G
\end{cases}
\]

\[
\text{mem}\{G_{\text{AG}}^A\} \sim \text{PYP}(a^A, b^A, G_{\text{AG}}^A)
\]
\[ L^A(d) = \sum_{r \in R_G : A \rightarrow \text{root}(\hat{d}_i), \ldots, \text{root}(\hat{d}_k)} \theta_r \prod_{i=1}^{k} \left[ \nu_{\hat{d}_i} G^\text{root}(\hat{d}_i) (\hat{d}_i) + (1 - \nu_{\hat{d}_i})1 \right] \]

\[
G^a_{FG}(d) = \begin{cases} 
\sum_{s \in \text{prefix}(d)} \text{mem}\{L^a\}(s) \prod_{i=1}^{n} G^\text{root}(s'_i) (s'_i) & \text{root}(d) = a \in V_G \\
1 & \text{root}(d) = a \in T_G 
\end{cases}
\]

\[
\text{mem}\{L^A\} \sim \text{PYP}(a^A, b^A, L^A)
\]
Fragment Grammars via Probabilistic Programming

1. Stochastic computation via unfold

2. Stochastic reuse via memoization

3. Partial computations via stochastic laziness
Context Free Grammars

\[
\begin{align*}
W & \rightarrow N \\
W & \rightarrow V \\
W & \rightarrow \text{Adj} \\
W & \rightarrow \text{Adv} \\
N & \rightarrow \text{Adj} -\text{ness} \\
N & \rightarrow \text{Adj} -\text{ity} \\
N & \rightarrow \text{electro-} N \\
N & \rightarrow \text{magnet} \\
N & \rightarrow \text{dog} \\
V & \rightarrow N -\text{ify} \\
V & \rightarrow \text{Adj} -\text{ize} \\
V & \rightarrow \text{re-} V \\
V & \rightarrow \text{agree} \\
V & \rightarrow \text{count} \\
\text{Adj} & \rightarrow \text{dis-} \text{Adj} \\
\text{Adj} & \rightarrow V -\text{able} \\
\text{Adj} & \rightarrow N -\text{ic} \\
\text{Adj} & \rightarrow N -\text{al} \\
\text{Adj} & \rightarrow \text{iall} \\
\text{Adv} & \rightarrow \text{Adj} -\text{ly} \\
\text{Adv} & \rightarrow \text{today} \\
\end{align*}
\]
Declarative Knowledge of Constituent Structure

\[\begin{align*}
&P_{v_1} \quad W \rightarrow N \\
&P_{v_2} \quad W \rightarrow V \\
&P_{v_3} \quad W \rightarrow Adj \\
&P_{v_4} \quad W \rightarrow Adv \\
&P_{a_1} \quad N \rightarrow Adj \quad -ness \\
&P_{a_2} \quad N \rightarrow Adj \quad -ity \\
&P_{a_3} \quad N \rightarrow electro- \quad N \\
&P_{a_4} \quad N \rightarrow magnet \\
&P_{a_5} \quad N \rightarrow dog \\
&P_{v_1} \quad V \rightarrow N \quad -ify \\
&P_{v_2} \quad V \rightarrow Adj \quad -ize \\
&P_{v_3} \quad V \rightarrow re- \quad V \\
&P_{v_4} \quad V \rightarrow agree \\
&P_{v_5} \quad V \rightarrow count \\
&P_{a_1} \quad Adj \rightarrow dis- \quad Adj \\
&P_{a_2} \quad Adj \rightarrow V \quad -able \\
&P_{a_3} \quad Adj \rightarrow N \quad -ic \\
&P_{a_4} \quad Adj \rightarrow N \quad -al \\
&P_{a_5} \quad Adj \rightarrow tall \\
&P_{adv_1} \quad Adv \rightarrow Adj \quad -ly \\
&P_{adv_2} \quad Adv \rightarrow today \\
\end{align*}\]
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`

```scheme
(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

\[(\text{unfold} \ 'N)\]

\[
\text{(define unfold}
  \text{(lambda (symbol)}
    \text{(if (terminal? symbol)}
      \text{symbol)
    \text{symbol)
    (map unfold (sample-rhs symbol))})
\]
Computation Trace

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

```
(unfold 'N)
```

```
(sample-rhs 'N)
```
Computation Trace

(unfold 'N)

(sample-rhs 'N)
Computation Trace

(unfold 'N)

(sample-rhs 'N)  N → Adj-ity
Computation Trace

\[ (\text{unfold 'N}) \]
\[ (\text{sample-rhs 'N}) \]
Computation Trace

(unfold 'N)

(sample-rhs 'N)

(unfold 'Adj)  (unfold 'ity)
Computation Trace

```
(unfold 'N)
  /
(sample-rhs 'N)
  /
(unfold 'Adj)  (unfold 'ity)
  /
(sample-rhs 'Adj)  'ity
  /
(unfold 'V)  (unfold 'able)
  /
(sample-rhs 'V)  'able
  /
(unfold 'agree)
  /
'agree
```
Trace as Tree

\[
\text{(unfold 'N)}
\]

\[
\text{(sample-rhs 'N)}
\]

\[
\text{(unfold 'Adj)}
\]

\[
\text{(sample-rhs 'Adj)}
\]

\[
\text{(unfold 'V)}
\]

\[
\text{(sample-rhs 'V)}
\]

\[
\text{'agree}
\]

\[
\text{'agree}
\]

\[
\text{'able}
\]

\[
\text{'ity}
\]

\[
\text{Adj}
\]

\[
\text{N}
\]

\[
\text{-ity}
\]

\[
\text{agree}
\]

\[
\text{-able}
\]
Reusability for PCFGs

Wednesday, November 16, 2011
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
...

Wednesday, November 16, 2011
(define eye-color
  (mem (lambda (person)
    (if (flip 0.5) 'blue brown))))
Reuse through Memoization

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

Anywhere in the program where (eye-color 'bob) is used, we will reuse same value.
(define eye-color
 (mem (lambda (person)
 (if (flip 0.5) 'blue 'brown))))

(eye-color 'bob) => 'blue
Reuse through Memoization

(define \texttt{eye-color}
  (mem (lambda (person)
         (if (flip 0.5) 'blue 'brown))))

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

\texttt{(eye-color 'bob)} => 'blue
\texttt{(eye-color 'bob)} => 'blue
Reusing through Memoization

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

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

```
(eye-color 'bob) => 'blue
(eye-color 'bob) => 'blue
(eye-color 'bob) => 'blue
```
(define eye-color
  (mem (lambda (person)
    (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
(eye-color 'bob)  => 'blue
Reuse through Memoization

```
(define eye-color
  (mem (lambda (person)
        (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
(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
(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
...

Wednesday, November 16, 2011
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
...

Wednesday, November 16, 2011
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:
  - $a \in [0, 1]$
  - $b > -a$

Probability of Reuse

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

$y_i$: Total number of observations of value $i$
Pitman-Yor Process

• Generalization of the Chinese Restaurant Process

• Two parameters:
  • \( a \in [0, 1] \)
  • \( b > -a \)

Probability of Reuse

\[
\frac{y_i - a}{N + b}
\]

\( y_i \): Total number of observations of value \( i \)

\( N \): Total number of observations
Pitman-Yor Process

- Generalization of the Chinese Restaurant Process

- Two parameters:
  - $a \in [0,1]$
  - $b > -a$

Probability of Reuse
$$\frac{y_i - a}{N + b}$$

Probability of Novelty
$$\frac{a \cdot K + b}{N + b}$$

$y_i$: Total number of observations of value $i$

$N$: Total number of observations
Pitman-Yor Process

- Generalization of the Chinese Restaurant Process

- Two parameters:
  - $a \in [0, 1]$
  - $b > -a$

Probability of Reuse
\[
\frac{y_i - a}{N + b}
\]

Probability of Novelty
\[
\frac{a \cdot K + b}{N + b}
\]

$y_i$: Total number of observations of value $i$
$N$: Total number of observations
$K$: Total number of values
(func arg1 ... argN)
\((\text{PYMem } a \ b \ \text{func})\)
$N=0$
$K=0$

\begin{center}
\begin{tabular}{c}
N=0 \\
K=0 \\
1 \\
\end{tabular}
\end{center}

\begin{center}
\begin{tabular}{c}
? \\
\ldots \\
\end{tabular}
\end{center}
$N=1$

$K=0$

1
$N=1$

$K=0$

1
\( v_4 \sim (\text{func \ arg1 \ ...}) \)

N=1
K=0

\( v_4 \)

1
N=1
K=1

Samples: $v_4$
$N=1$  $K=1$

$\begin{array}{ccc}
1-a & a \cdot 1+b \\
1+b & 1+b
\end{array}$

Samples: $v_4$
\[ y_i - a \]
\[ N + b \]

\[ \frac{1-a}{1+b} \quad \frac{a \cdot 1+b}{1+b} \]

Samples:  \( v_4 \)
\[
\begin{align*}
\frac{y_i - a}{N + b} & \quad \frac{a \cdot K + b}{N + b} \\
\frac{1-a}{1+b} & \quad \frac{a \cdot 1+b}{1+b}
\end{align*}
\]

Samples: \( v_4 \)
\[
\frac{a \cdot K + b}{N + b}
\]

\[N = 1\]
\[K = 1\]

Samples: \(v_4\)
\[ N = 1 \quad K = 1 \]

\[
\begin{align*}
\text{v}_4 & \quad ? \quad \ldots \\
\frac{1{-}a}{1{+}b} & \quad \frac{a{\cdot}1{+}b}{1{+}b}
\end{align*}
\]

Samples: \( \text{v}_4 \)
$N = 1$

$K = 1$

$\begin{align*}
1 - a & \quad a \cdot 1 + b \\
\frac{1}{1+b} & \quad \frac{a}{1+b}
\end{align*}$

Samples: $v_4$
\[ \frac{1-a}{1+b}, \frac{a\cdot 1+b}{1+b} \]

Samples: \( v_4 \)
$N = 2$

$K = 1$

\[ \frac{1-a}{1+b} \quad \frac{a \cdot 1 + b}{1+b} \]

Samples: $v_4$
\( v_1 \sim (\text{func arg1 ...}) \)

\[
\begin{array}{ccc}
N=2 & v_4 & v_1 \\
K=1 & \frac{1-a}{1+b} & \frac{a \cdot 1+b}{1+b} \\
\end{array}
\]

Samples: \( v_4 \)
\( \text{Samples: } v_4, v_1 \)
\[ y_i - a \]
\[ \frac{1}{N + b} \]
\[ N = 2 \]
\[ K = 2 \]

\[ a \cdot K + b \]
\[ \frac{1 - a}{2 + b} \]
\[ 1 - a \cdot 2 + b \]
\[ 2 + b \]

Samples: \( v_4, v_1 \)
\[
\frac{a \cdot K + b}{N + b}
\]

\[
\frac{1 - a}{2 + b} \quad \frac{1 - a}{2 + b} \quad \frac{a \cdot 2 + b}{2 + b}
\]

Samples: \(v_4, v_1\)
N=2
K=2

Samples: \( v_4, v_1 \)
$N=2$
$K=2$

Samples: $v_4, v_1$
$N=3$
$K=2$

Samples: $v_4, v_1, v_4$

\[
\frac{1 - a}{2 + b} \quad \frac{1 - a}{2 + b} \quad \frac{a \cdot 2 + b}{2 + b}
\]
\[
\begin{align*}
y_i - a & \quad \frac{a \cdot K + b}{N + b} \\
\frac{N}{N + b} & \quad \frac{2 - a}{3 + b} \quad \frac{1 - a}{3 + b} \quad \frac{a \cdot 2 + b}{3 + b}
\end{align*}
\]

Samples: \(v_4, v_1, v_4\)
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.
(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).

\[
\begin{array}{c}
N \\
\text{Adj} \quad -\text{ity} \\
\text{V} \quad -\text{able} \\
\text{agree}
\end{array}
\]
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).

\[
\text{N} \quad \text{Adj} \quad -\text{ity} \\
\quad \text{V} \quad -\text{able} \\
\quad \text{agree}
\]

\[
\text{N} \quad \text{Adj} \quad -\text{ity} \\
\quad \text{V} \quad -\text{able}
\]
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

N

Adj -ity

V -able
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

\((\text{add3} \ (\ + \ 1 \ 2 \ 3) \ (\ * \ 2 \ 4) \ (\ - \ 3 \ 1))\)
Eager Evaluation

(add3 (+ 1 2 3) (* 2 4) (− 3 1))
Eager Evaluation

\[(\text{add3} \ 6 \ (* \ 2 \ 4) \ (- \ 3 \ 1))\]
Eager Evaluation

\[(\text{add3} \ 6 \ (* \ 2 \ 4) \ (- \ 3 \ 1))\]
Eager Evaluation

\[(\text{add3} \ 6 \ 8 \ (- \ 3 \ 1))\]
Eager Evaluation

\((\text{add3} \ 6 \ 8 \ (- \ 3 \ 1))\)
Eager Evaluation

\((\text{add3} \ 6 \ 8 \ 2)\)
Eager Evaluation

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

(add3 6 8 2)
Eager Evaluation

16
Lazy Evaluation

\((\text{add3} \ ( + \ 1 \ 2 \ 3) \ (* \ 2 \ 4) \ (- \ 3 \ 1))\)
Lazy Evaluation

\[
(\text{add3}\ ((+\ 1\ 2\ 3)\ (*\ 2\ 4)\ (\ -\ 3\ 1)))
\]

\[
(\text{define}\ \text{add3}\n\ (\text{lambda}\ (x\ y\ z)\n\ (+\ x\ y\ z)))
\]
Lazy Evaluation

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

\[
\begin{align*}
\text{Lazy Evaluation} & \quad \quad (\text{define add3}) \\
\text{(lambda \((x \ y \ z)\))} & \quad \quad (+ \ x \ y \ z))
\end{align*}
\]

\[
\begin{align*}
(+ & \quad (+ \ 1 \ 2 \ 3) \quad (* \ 2 \ 4) \quad (- \ 3 \ 1)) \\
\text{x} & \quad \quad \text{y} \quad \quad \text{z}
\end{align*}
\]
Lazy Evaluation

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
\( \lambda \)-calculus: Order of Evaluation
\(\lambda\)-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.
\(\lambda\)-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 \(\lambda\)-calculus.
\( \lambda \)-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 \( \lambda \)-calculus.
- Does matter for \( \Psi \lambda \)-calculus!
\[\lambda\text{-calculus: Order of Evaluation}\\
\]

\[
(define \text{same?}\\
(lambda (x)\\
 (equal? x x)))\\
\]
Ψλ-calculus: Order of Evaluation

(define same? (lambda (x) (equal? x x)))
$\Psi\lambda$-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)))

P(true) = 1

eager

(same? (flip))
\( \Psi \lambda \)-calculus: Order of Evaluation

\[
\text{(define same?}
\begin{align*}
& (\text{lambda } (x) \\
& \quad (\text{equal? } x x))
\end{align*}
\text{)}
\]

\text{eager } \quad P(\text{true}) = 1

\text{lazy } \quad P(\text{true}) = \frac{1}{2}

\text{(same? } (\text{flip}))
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 $\Psi\Lambda$-calculus.
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.
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.

• Assume eager evaluation strategy and add delay primitive.
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.

• 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

```
(unfold 'N)
 |
(sample-rhs 'N)
 |
(dealy-or-unfold 'Adj) (delay-or-unfold 'ity)
 |
(unfold 'Adj) (unfold 'ity)
 |
(sample-rhs 'Adj) 'ity
 |
(dealy-or-unfold 'V) (dealy-or-unfold 'able)
 |
(delay (unfold 'V)) (unfold 'able)
 |
'able
```
Computation Trace with Delay

\[
\begin{align*}
\text{(unfold 'N)} & \\
| & \\
\text{(sample-rhs 'N)} & \\
| & \\
\text{(dealy-or-unfold 'Adj)} & \text{(delay-or-unfold 'ity)} \\
| & | \\
\text{(unfold 'Adj)} & \text{(unfold 'ity)} \\
| & | \\
\text{(sample-rhs 'Adj)} & 'ity \\
| & | \\
\text{(dealy-or-unfold 'V)} & \text{(dealy-or-unfold 'able)} \\
| & | \\
\text{(delay (unfold 'V))} & \text{(unfold 'able)} \\
| & | \\
'able \\
\end{align*}
\]
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
1. Always possible to use base grammar.
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)
- Store whole form after first use.
- 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)
- Store best set of subcomputations for explaining the data.
- Generalization of *Adaptor Grammars*
Outline

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

<table>
<thead>
<tr>
<th>Productive</th>
<th>+ness (goodness), +ly (quickly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-productive</td>
<td>+ity (ability), +or (operator)</td>
</tr>
<tr>
<td>Unproductive</td>
<td>+th (width)</td>
</tr>
</tbody>
</table>

- **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

```
N
  Adj
    V
      agree

N
  V
    V
      affirm

N
  V
    V
      able

N
  V
    V
      ity

N
  V
    V
      ione

N
  V
    V
      ate
```
# English Derivational Morphology

<table>
<thead>
<tr>
<th>Productive</th>
<th>Semi-productive</th>
<th>Unproductive</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ness ( (goodness) ), +ly ( (quickly) )</td>
<td>+ity ( (ability) ), +or ( (operator) )</td>
<td>+th ( (width) )</td>
</tr>
</tbody>
</table>

1. Individual suffix productivity differences \(-ness/-ity/-th\).
2. Suffix sequences.
### English Derivational Morphology

#### 1. Individual suffix productivity differences (-ness/-ity/-th).

<table>
<thead>
<tr>
<th>Productive</th>
<th>+ness (goodness), +ly (quickly)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semi-productive</td>
<td>+ity (ability), +or (operator)</td>
</tr>
<tr>
<td>Unproductive</td>
<td>+th (width)</td>
</tr>
</tbody>
</table>

#### 2. Suffix combinations.
Productivity

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

- Analyses.
  1. Examples of highly productive affixes.
  2. Convergence with other theoretical measures.
How is Productivity Represented?

- Relative probability of fragments with or without variables.
Productivity Analyses

1. Examples of highly productive suffixes.

2. Convergence with other theoretical measures.
Top 5 Most Productive Suffixes

**MDPCFG** *(Full-Parsing)*

- **Suffix**
  - ion:V>N
  - ly:Adj>Adv
  - ate:BND>V
  - ment:V>N
  - er:V>N
- **Example**
  - regression
  - quickly
  - segregate
  - development
  - talker

**MAG** *(Full-listing)*

- **Suffix**
  - ly:Adj>Adv
  - ion:V>N
  - er:V>N
  - y:N>Adj
  - ly:V>Adv
- **Example**
  - quickly
  - regression
  - talker
  - bitingly
  - mousey

**FG** *(Inference-based)*

- **Suffix**
  - ly:Adj>Adv
  - er:V>N
  - ness:Adj>N
  - y:N>Adj
- **Example**
  - quickly
  - talker
  - tallness
  - mousey
  - prisoner

**DOPI** *(Exemplar)*

- **Suffix**
  - ion:V>N
  - er:V>N
  - ment:V>N
  - ate:BND>V
  - ly:Adj>Adv
- **Example**
  - regression
  - talker
  - development
  - segregate
  - quickly

**GDMN** *(Exemplar)*

- **Suffix**
  - ion:V>N
  - ly:Adj>Adv
  - ment:V>N
  - er:V>N
  - ate:BND>V
- **Example**
  - regression
  - quickly
  - development
  - segregate
  - talker
Top 5 Most Productive Suffixes

MDPCFG (Full-Parsing)

- Example: regression, quickly, segregate, development, talker

MAG (Full-listing)

- Example: quickly, regression, talker, bitingly, mousey

FG (Inference-based)

- Suffix: ly:Adj>Adv, er:V>N
- Example: quickly, talker
- Example: ness:Adj>N, tallness

GDMN (Exemplar)

- Example: regression, talker, development, segregate, quickly

DOP1 (Exemplar)

- Example: regression, talker, development, segregate, quickly

mousey, prisoner
Top 5 Most Productive Suffixes

**MDPCFG** *(Full-Parsing)*

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ion:V&gt;N</td>
<td>regression</td>
</tr>
<tr>
<td>ly:Adj&gt;Adv</td>
<td>quickly</td>
</tr>
<tr>
<td>ate:BND&gt;V</td>
<td>segregate</td>
</tr>
<tr>
<td>ment:V&gt;N</td>
<td>development</td>
</tr>
<tr>
<td>er:V&gt;N</td>
<td>talker</td>
</tr>
</tbody>
</table>

**MAG** *(Full-listing)*

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ion:V&gt;N</td>
<td>regression</td>
</tr>
<tr>
<td>ly:Adj&gt;Adv</td>
<td>quickly</td>
</tr>
<tr>
<td>er:V&gt;N</td>
<td>talker</td>
</tr>
<tr>
<td>ly:V&gt;Adv</td>
<td>bitingly</td>
</tr>
<tr>
<td>y:N&gt;Adj</td>
<td>mousey</td>
</tr>
</tbody>
</table>

**FG** *(Inference-based)*

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ly:Adj&gt;Adv</td>
<td>quickly</td>
</tr>
<tr>
<td>er:V&gt;N</td>
<td>talker</td>
</tr>
<tr>
<td>ness:Adj&gt;N</td>
<td>tallness</td>
</tr>
<tr>
<td>y:N&gt;Adj</td>
<td>mousey</td>
</tr>
<tr>
<td>er:N&gt;N</td>
<td>prisoner</td>
</tr>
</tbody>
</table>

**DOP1** *(Exemplar)*

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ion:V&gt;N</td>
<td>regression</td>
</tr>
<tr>
<td>er:V&gt;N</td>
<td>talker</td>
</tr>
<tr>
<td>ment:V&gt;N</td>
<td>development</td>
</tr>
<tr>
<td>ate:BND&gt;V</td>
<td>segregate</td>
</tr>
<tr>
<td>ly:Adj&gt;Adv</td>
<td>quickly</td>
</tr>
</tbody>
</table>

**GDMN** *(Exemplar)*

<table>
<thead>
<tr>
<th>Suffix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>ion:V&gt;N</td>
<td>regression</td>
</tr>
<tr>
<td>ly:Adj&gt;Adv</td>
<td>quickly</td>
</tr>
<tr>
<td>ment:V&gt;N</td>
<td>development</td>
</tr>
<tr>
<td>er:V&gt;N</td>
<td>talker</td>
</tr>
<tr>
<td>ate:BND&gt;V</td>
<td>segregate</td>
</tr>
</tbody>
</table>
Productivity Analyses

1. Examples of highly productive suffixes.

2. Convergence with other theoretical measures.
Baayen’s Corpus-Based Measures

• Baayen’s $\mathcal{P} / \mathcal{P}^*$ (e.g., Baayen, 1992)

• $\mathcal{P}$: $\text{Prob(NOVEL | SUFFIX)}$ i.e. rate of growth of forms with suffix

• $\mathcal{P}^*$: $\text{Prob(SUFFIX | NOVEL)}$ i.e. rate of growth of vocabulary due to suffix
# Productivity Correlations

(*\(P/P^*\) values from Hay & Baayen, 2002)

<table>
<thead>
<tr>
<th>Model</th>
<th>FG (Inference-based)</th>
<th>MDPCFG (Full-parsing)</th>
<th>MAG (Full-listing)</th>
<th>DOPI (Exemplar-based)</th>
<th>GDMN (Exemplar-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P)</td>
<td>0.907</td>
<td>-0.0003</td>
<td>0.692</td>
<td>0.346</td>
<td>0.143</td>
</tr>
<tr>
<td>(P^*)</td>
<td>0.662</td>
<td>0.480</td>
<td>0.568</td>
<td>0.402</td>
<td>0.500</td>
</tr>
</tbody>
</table>
# English Derivational Morphology

<table>
<thead>
<tr>
<th>Type</th>
<th>Suffixes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Productive</strong></td>
<td>+<strong>ness</strong> (goodness), +<strong>ly</strong> (quickly)</td>
</tr>
<tr>
<td><strong>Semi-productive</strong></td>
<td>+<strong>ity</strong> (ability), +<strong>or</strong> (operator)</td>
</tr>
<tr>
<td><strong>Unproductive</strong></td>
<td>+<strong>th</strong> (width)</td>
</tr>
</tbody>
</table>

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

2. Suffix combinations.
Suffix Combinations

1. Suffix Ordering.

2. Generalization of Suffix Combinations.
Suffix Combinations

1. Suffix Ordering.

2. Generalization of Suffix Combinations.
Suffix Ordering

- Derivational morphology hierarchical and recursive.
- Multiple suffixes can appear in a word.
Suffix Combinations
Suffix Combinations

- Many, many combinations of suffixes do not appear in words (even taking into account categories).
Suffix Combinations

• Many, many combinations of suffixes do not appear in words (even taking into account categories).

• Fabb (1988).
Suffix Combinations

• Many, many combinations of suffixes do not appear in words (even taking into account categories).

• Fabb (1988).
  - 43 suffixes.
Suffix Combinations

• Many, many combinations of suffixes do not appear in words (even taking into account categories).

• Fabb (1988).
  - 43 suffixes.
  - 663 possible pairs.
Suffix Combinations

• Many, many combinations of suffixes do not appear in words (even taking into account categories).

• Fabb (1988).
  - 43 suffixes.
  - 663 possible pairs.
  - Only 50 exist.
Complexity-Based Ordering  (Hay, 2002)

On average, more productive suffixes appear after (outside of) less productive suffixes.
Measuring Ordering

• Examine attested orderings in corpus.

• *Mean rank* of each affix (Plag and Baayen, 2009).
  • Graph-theoretic statistic.
  • Measures degree to which each suffix tends to occur after other suffixes (on average).

• Compute *log odds* of suffix appearing second versus first for each model.
<table>
<thead>
<tr>
<th>Model</th>
<th>FG (Inference-based)</th>
<th>MDPCFG (Full-parsing)</th>
<th>MAG (Full-listing)</th>
<th>DOPI (Exemplar-based)</th>
<th>GDMN (Exemplar-based)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean Rank</td>
<td>0.568</td>
<td>0.275</td>
<td>0.424</td>
<td>0.452</td>
<td>0.431</td>
</tr>
</tbody>
</table>
Suffix Combinations

1. Suffix Ordering.

2. Generalization of Suffix Combinations.
Generalizable Combinations

Frozen Combinations

Generalizable Combinations

affirm -ate

V

N

V

-ate

-ion

Adj

N

V

-able

-ity
Generalizable Combinations

Frozen Combinations

Generalizable Combinations

affirm

V

-ate

N

-V

-ion

Adj

N

-ity

V

-able
-ity v. -ness

-ness more productive than -ity.

-ity more productive than -ness after: -ile, -able, -(i)an, -ic.

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).
  - remortibility > remortibleness.
-ivity v. -bility

-ive

-ble

-ness

-ity

Predicted

0

ble ive
-ivity v. -bility

-ive
-ble

-ness

-ity

Preference for -ness

Predicted

Preference for -ness

146
-ivity v. -bility

-ive
-ble

-ness

-ity

Preference for -ity

Predicted

}{

{

{

{

{

{0, 5}

{0, 5, 10}

{}
-ivity v. -bility

-ive
-ble

-ness

-ity

Predicted

Preceding suffix -ive
-ivity v. -bility

-ive

-ble

-ness

-ity

Preceding suffix -ble
(Exemplar-based)
-ive
-ble
-ness

-ity

Predicted
MDPCFG (Full-parsing)
MAG (Full-listing)
DOPI (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!