Computer Science and Artificial Intelligence Laboratory
Technical Report

MIT-CSAIL-TR-2009-013 March 31, 2009

Fragment Grammars: Exploring Computation and Reuse in Language

Timothy J. O'Donnell, Noah D. Goodman, and Joshua B. Tenenbaum

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Fragment Grammars: Exploring Computation and Reuse in Language

Timothy J. O'Donnell (timo@wjh.harvard.edu)
Harvard University Department of Psychology

Noah D. Goodman
MIT, Brain and Cognitive Science

Joshua B. Tenenbaum
MIT, Brain and Cognitive Science

Contents

Abstract

Language relies on a division of labor between stored units and structure building operations which combine the stored units into larger structures. This division of labor leads to a tradeoff: more structure-building means less need to store while more storage means less need to compute structure. We develop a hierarchical Bayesian model called fragment grammar to explore the optimum balance between structure-building and reuse. The model is developed in the context of stochastic functional programming (SFP), and in particular, using a probabilistic variant of Lisp known as the Church programming language [17]. We show how to formalize several probabilistic models of language structure using Church, and how fragment grammar generalizes one of them—adaptor grammars [21]. We conclude with experimental data with adults and preliminary evaluations of the model on natural language corpus data.

1 Introduction and Overview

Perhaps the most celebrated feature of human language is its productivity. Language allows us to express and comprehend an unbounded number of thoughts using only finite resources. This productivity is made possible by a fundamental design feature of language: a division of labor between stored units (such as words) and structure building computations which combine these stored units into larger representations.

Although all theories of language make some distinction between what is stored and what is computed, they vary widely in the way in which they actually implement the interaction. Among the many disagreements are questions about what kinds of things can be stored, what conditions cause something to be stored (or not), and how storage might be integrated with computation. It is beyond the scope of the present report to review the relevant linguistic and psycholinguistic literatures on storage versus computation (see e.g. [30]), instead we will offer a new model of this problem known as *fragment grammar*.

Fragment grammars formalize the intuitive idea that there is a tradeoff between storage and computation. More computation means less need to store while more storage means less need to compute. Using tools from hierarchical Bayesian statistics, we will show how this balance can be optimized.

Fragment grammars are a generalization of the *adaptor grammar* model introduced by [21]. Unlike adaptor grammars, however, fragment grammars are able to store *partial* as well as complete computations.

This report takes a non-traditional approach to the presentation of the modeling work. We will use the paradigm of *stochastic functional programming* (SFP) and in particular the Church programming language [17] as a notation to formalize the model. Church is a stochastic version of the lambda calculus with a Scheme-like syntax built on a sampling semantics. The aim of Church, and of SFP in general, is to develop rich compositional languages for expressing probabilistic models and to provide generalized inference for those models.[1]

Church is aimed at allowing the compact expression of complex hierarchical Bayesian generative models. We use Church to describe the fragment grammar model for three reasons. First, Church excels at concisely, but accurately, expressing models involving recursion. Recursion is at the core of linguistic models; however, most current frameworks for probabilistic modeling, such as graphical models, make it difficult or impossible to express.

Second, Church allows us to develop models compositionally, reusing parts where appropriate. In this report, rather than immediately formalizing frag-

[1] We do not make use of Church's generalized inference engine, which is not currently effective for linguistic representations. This is an area of active research.

ment grammars, we will start by formalizing probabilistic context-free grammars using Church. From this starting point, we will then generalize the model to adaptor grammars, two-stage adaptor grammars, and finally to fragment grammars. By expressing each stage in Church, the exact relationships between the models will be completely explicit, highlighting the reused parts and the innovations.

The final reason that we choose to use Church is the language-level support it provides for *reuse* of computation in models. Our analysis of the storage versus computation problem relies crucially on the notion of reuse. Church provides constructs for this in the language itself. We discuss this in more detail below.

1.1 Linguistic and Psychological Motivation

A central question of the psychology of language and linguistics is what constitutes the *lexicon*. Here we are using the term 'lexicon' to to refer to the set of stored units, of whatever form, used to produce or comprehend an utterance.[2]

Traditional approaches have tended to view the lexicon as consisting of just word or morpheme-sized units [5]. However, the past twenty years have seen the emergence of theories which advocate a *heterogeneous lexicon* (e.g. [19, 7, 34, 6, 14, 32], amongst many others).

In a heterogeneous approach to the lexicon, the idea of a lexical item—a stored piece of structure—is divorced from particular linguistic categories such as *word* or *morpheme*. These categories still exist, to be sure, but they are morphological, phonological, or syntactic constructs, independent of the question of storage. The divorce of categorical structure from storage means a much wider variety of items can be stored in the mental lexicon. These include structures both smaller and larger than words.

It is beyond the scope of this report to discuss the many empirical arguments in favor of the heterogeneous approach to lexical storage. Interested readers are directed to any of the citations above. The approach does, however, raise several important questions:

1. Where does the inventory of lexical items come from in the first place? That is, how is it learned?
2. How is the process of using and creating stored lexical items integrated with the process of generating linguistic structures?

[2]In this report we will use the term *lexicon* to specifically refer to this repository of stored fragments of structure. We will not be concerned with other senses of the term, such as those used in theories of what constitutes a morphological or phonological word. These stored units are sometimes referred to as *listemes* [8].

To answer these questions we will first re-formulate the storage/computation problem. Rather than viewing storage as an end in itself, we will hypothesize that instead, storage is just the result of attempting to reuse computational work whenever it makes sense to do so. In other words, we propose that during the production and comprehension of language, the language faculty will try to reuse work done previously. In order to reuse a previous computation, that computation must be stored in memory. Under this view, the goal is not storage, but rather the optimal reuse of previously computed structure.

This perspective leads to an immediate question facing the language user: what is the optimal set of computations to store as lexical items for my language? Working in the Bayesian framework, this can be recast as a question of posterior inference: what is the posterior distribution over lexica given some input data.

1.2 Computation versus Reuse as a Bayesian trade-off

As discussed above, our fundamental Bayesian question is this: given some data, what should the lexicon look like? We will now illustrate the trade-off inherent in this question.

Imagine that we have some generative process producing syntactic trees.[3] A few trees created by this generative process—either during production or comprehension—are shown in Figure 1. Each row shows the consequences for reuse and reusability of three different approaches to the storage of previously computed structure. In each row, the fragments of the trees highlighted in the same color are the fragments which are reusable *in this set of trees*.

This figure shows two extreme kinds of lexical item storage, and an intermediate case. The first row shows the consequences of only storing **minimal** sized lexical items. In this case, the items all correspond to depth-one trees. When we store very small, abstract fragments such as these, they have a high degree of reusability. They can appear in many different syntactic trees. We can reuse them easily when producing or comprehending an utterance. However, many independent choices are required to build any single expression. In a probabilistic setting, independent choices correspond to computational work, and therefore if we only store these minimal sized fragments, we will have to do more work each time we build an utterance.

[3] For example, a probabilistic context-free grammar of the kind we will discuss below. In this report, we will use examples drawn mostly from syntax. The model, however, is meant to be a general model of reuse in hierarchical generative processes and can be applied easily to problems in phonology, morphology, and semantics.

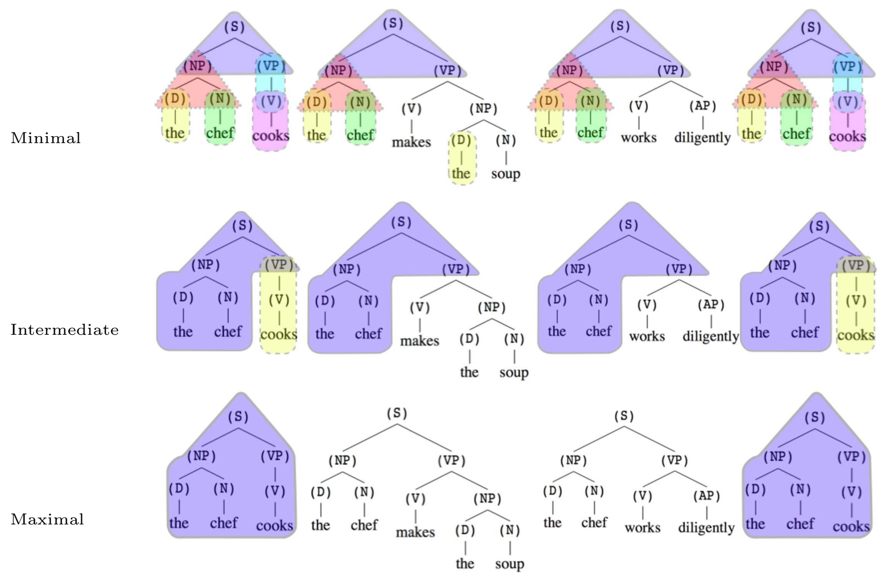

Figure 1: This figure shows the consequences for reuse of three possible storage regimes. **Minimal** sized fragments allow fine-grained reuse, but force many choices when generating a sentence. **Maximal** sized fragments of structure allow fewer choices to generate a sentence, but limit reusability and therefore increase the number of fragments. Storage of an **intermediate** size optimizes this balance.

The third row shows the consequences of storing **maximal** sized lexical items. Maximal sized fragments correspond to entire syntactic trees. In other words, they correspond to storing all utterances produced or comprehended in their entirety every time. In this setting, if we have already seen a particular structure, we can generate it again by making only a single independent choice and reusing the whoe stored structure. In general, as lexical items grow bigger we will require fewer of them to generate any single expression. The amount of computation done per tree will be minimized. However, any single lexical item can be used fewer times across expressions. In fact, we will only be able to use a stored lexical item if we want to reproduce an entire utterance exactly in the same way in the future. As a result, we will have to store many more fragments in memory.

Better solutions fall in-between these two extremes. They simultaneously optimize the number of choices that must be made to generate a set of expressions and the number of lexical items that must be stored. This is illustrated in the middle row of figure 1, where **intermediate** sized fragments of structure are stored. The fragments in this row allow more reuse than the third row with fewer independent choices per tree than the first row.

The fragment grammar model defines a probability distribution over the set of different ways of making the reuse/computation tradeoff. We will show below how Bayesian inference can be used to optimize this tradeoff.

1.3 Lexical Items as Distributions in a Two-stage Generative Process

We have emphasized reuse as the organizing principle behind our model of the lexicon. Below we will formalize reuse in generative processes via *stochastic memoization.* Memoization is a technique long-known in computer science whereby the outputs of computations are stored in memory so that they can be reused by later computations. Stochastic memoization lifts this idea to the probabilistic case. We will describe both below.

Stochastic memoization as embodied in Church provides a bridge between ideas from non-parametric Bayesian statistics and ideas from functional programming. In this report, we will show how to use stochastic functional programs with stochastic memoization to elegantly express complex, recursive, non-parametric Bayesian models of language. In particular, we will show how stochastic memoization can lead to an elegant presentation of adaptor grammars.

As we mentioned earlier, fragment grammars generalize the adaptor grammar model. This generalization relies on two novel ideas which constitute the main technical innovations of the present report: lexical items as distributions, and a two-stage generative process.

Typically, in generative models of language, lexical items are thought of as static structures which are assembled into a linguistic expression. We will adopt another perspective in which lexical items are active entities that themselves can construct linguistic expressions—distributions.

If a lexical item is a distribution how do we construct an expression? We will show how this can be done using a two-stage generative process. First, choose a (perhaps novel) lexical item; second, sample this lexical item to get a linguistic expression. Because lexical items are defined recursively, this second stage will often involve choosing another lexical item. Thus, our basic generative process, will alternate between sampling lexical items and sampling expressions from those lexical items.

In such a two-stage model, there is no distinction between processes which comprehend and produce language and processes which learn the lexicon. The same model which provides for language use also provides for language learning.

1.4 Outline

Our plan for the rest of the report is as follows. First, we introduce some basic notions from stochastic functional programming and show how a well-known model of language structure, the probabilistic context-free grammar (PCFG), can be formulated in these terms. We then introduce stochastic memoization, discuss its relation to certain tools from non-parametric Bayesian statistics, and show how the adaptor grammar model can be formulated in these terms.

We then show how the adaptor grammar model can be reformulated as a two-stage model with lexical-items as procedures. This leads directly to the definition of fragment grammars. Having defined fragment grammars, we move on to discuss inference in the model, taking some time to discuss intuitions about when fragment grammars store lexical items, and when they do not. Finally, we report some preliminary data to evaluate the fragment grammar model. In the appendices we discuss several points of mathematical interest.

Throughout this report, we use a mixture of verbal description, Church code, and mathematical notation to describe the various models. Most mathematical notation is included only for precision and can be skipped by the reader on a first pass.

2 Generative Processes as Random Procedures

In SFP, generative processes are defined using *random procedures*. A traditional deterministic procedure[4] always returns the same value when applied to the same arguments. A random procedure, on the other hand, defines a distribution over outputs given inputs.

A familiar example of a random procedure is the `rand` or `random` function of most programming languages. `rand` typically returns a value on the interval $[0, 1]$ with uniform probability. Another example of a random procedure is `flip`. This procedure takes a weight as input and flips a biased coin according to the weight, returning `true` or `false` accordingly.

With an inventory of such basic *elementary random procedures* (ERPs), more complex procedures may be constructed. A simple example of such a procedure is expressed in Church code in Figure 2. This code defines a function called `noisy-or`.[5] With probability ϵ, `noisy-or` simply returns the result of applying `or` to its arguments. However, with probability $1 - \epsilon$, it returns the opposite of that result (if `(or arguments)` is `true` it returns `false`, and vice versa).

```
(define noisy-or
  (lambda arguments
    (if (flip ε)
        (or arguments)
        (not (or arguments)))))
```

Figure 2: Church code defining a noisy-or function.

The probability distribution defined by this and other complex procedures is the product of all the ERPs which are evaluated in the course of invoking the procedure. For example, if we applied `noisy-or` to the arguments `0` and `1`: `(noisy-or 0 1)`, then the probability of the outcome `true` would be ϵ and the

[4]Procedures are often called "functions." Here we distinguish between functions, which are the mathematical objects which procedures compute, and procedures, which are instantiated programmatic recipes for computing them.

[5]The syntactic form `(lambda arguments body)` is an operator which constructs a procedure with arguments `arguments` and body `body`—this is the fundamental abstraction operator of the λ-calculus.

probability of `false` would be $1-\epsilon$. For a more complex example consider the following expression: `(noisy-or (flip 0.5) (flip 0.5))`. Now the probability of `true` depends both on the evaluation of the `flip`s determining the two arguments as well as the `flip` inside the procedure body.

The set theoretic meaning[6] of a procedure in Church is a stochastic function.[7] That is, it is a mapping from the procedure's inputs to a distribution over its outputs. If a procedure takes no arguments, then this mapping is constant, and the procedure defines a probability distribution. In functional programming, a procedure of no arguments is called a *thunk*. Therefore, in the stochastic setting thunks are identified with probability distributions. Another interpretation of a Church procedure is as a sampler.[8] Application of a procedure to some arguments causes it to sample a value from the distribution over outputs that corresponds to those arguments. For example, `noisy-or` can be thought of as a sampler which samples from the set {`true, false`} conditioned on its inputs.

2.1 Probabilistic Context-free Grammars as Random Procedures

In this section, we show how probabilistic context-free grammars can be formulated in a SFP setting. Context-free grammars (CFGs) are a simple, widely-known, and well-studied formalism for modeling hierarchical structure and computation [1].[9]

Formally, a context-free grammar is a 4-tuple, $G = \langle V, W, R, \mathtt{S} \rangle$ where

- V is a finite set of *nonterminal* symbols.
- W is the set of *terminal* symbols, pairwise disjoint from V.
- $R \subseteq V \times (V \cup W)^*$ is the set of *productions*, or *rules*.
- $\mathtt{S} \in V$ is a distinguished *start symbol*.

By convention nonterminals are written with capital letters and, when used to model syntactic structure, they represent categories of constituents such

[6] That is, the denotational semantics.

[7] Stochastic functions are sometimes called a probabilistic kernels.

[8] Sampling can be thought of as the operational semantics of a procedure in Church.

[9] CFGs and PCFGs are widely known in both linguistics and computational linguistics to be inadequate as models of natural language structure (see for example discussions in [23]). However, they are in some sense the simplest generative model which captures the idea of arbitrary (recursive) hierarchical structure. It is also the case that many other more sophisticated linguistic formalisms have context-free derivation trees even when their derived structures are non-context-free; one class of such systems are the *linear context-free rewrite systems* [37].

as “noun phrase” (`NP`) or “verb” (`V`). The unique, distinguished nonterminal known as the start symbol is written `S`. This symbol represents the category of complete derivations, or sentences. Terminals, written with lowercase letters, typically represent words or morphemes (e.g., “chef”, or “soup”).

The production rules, which are written $\mathtt{A} \longrightarrow \gamma$, where γ is some sequence of terminals and nonterminals, and `A` is a nonterminal, define the set of possible computations for the system. For example the rule $\mathtt{S} \longrightarrow \mathtt{NP}\ \mathtt{VP}$ says that a constituent of type `S` can be computed by first computing a noun phrase `NP` and a verb phrase `VP`, and then concatenating the results. The set of computations defined by a given CFG is hierarchical and can be recursive. The list of symbols to the right of the arrow is referred to as the *right-hand side* (RHS) of that production. The nonterminal to the left of the arrow is the rule’s *left-hand side* (LHS). Terminal symbols, such as ‘the’ or ‘works’ are atomic values which cause an expression-building recursion to stop and return. Starting with the start symbol `S` and following the rules until we only have terminals, it is possible to derive sequences of words such as ⟨the chef cooks the omelet⟩ or ⟨the chef works diligently⟩. A sequence of such computations can be represented by a *parse tree* like that on the right-hand side of Figure 3

`S`	$\longrightarrow$	`NP`	`VP`
`VP`	$\longrightarrow$	`V`	`NP`
`VP`	$\longrightarrow$	`V`	`AP`
`NP`	$\longrightarrow$	`D`	`N`
`AP`	$\longrightarrow$	`A`	
`D`	$\longrightarrow$	the	
`D`	$\longrightarrow$	a	
`N`	$\longrightarrow$	chef	
`N`	$\longrightarrow$	omelet	
`N`	$\longrightarrow$	soup	
`V`	$\longrightarrow$	cooks	
`V`	$\longrightarrow$	works	
`V`	$\longrightarrow$	makes	
`A`	$\longrightarrow$	diligently	

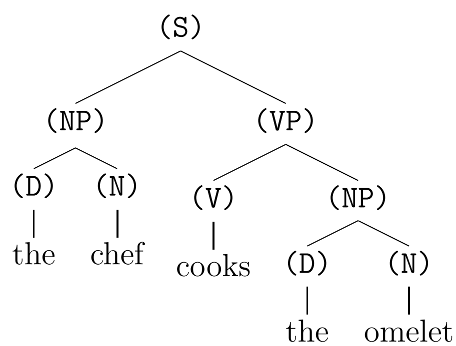

Figure 3: A simple context-free grammar and corresponding parse tree. As explained below, the parentheses represent procedure application.

A CFG encodes the possible choices that can be made in computing an expression, but does not specify how to make the choices. That is, the choices are *non-deterministic*. Like other non-deterministic generative systems CFGs also have a natural probabilistic formulation known as Probabilistic Context-Free Grammars (PCFGs). Making a context-free grammar probabilistic consists of replacing the non-deterministic choice with a random procedure which samples production RHSs from some distribution. There are various ways to define this distribution, but the simplest draws the possible RHSs of a production from a multinomial distribution.

Assume that we have an elementary random procedure called `multinomial` which is called like this: `(multinomial values probabilities)` where `values` is a list of objects and `probabilities` is a list of probabilities for each object. This procedure will return samples of objects from `values` according to the distribution in `probabilities`. We can define the generative process for a PCFG with the Church code in figure 4.

```
(define D (lambda ()
            (map sample
                 (multinomial
                  (list (terminal "the")
                        (terminal "a"))
                  (list θ1^D θ2^D)))))

(define N (lambda ()
            (map sample
                 (multinomial
                  (list (terminal "chef")
                        (terminal "soup")
                        (terminal "omelet"))
                  (list θ1^N θ2^N θ3^N)))))

(define V (lambda ()
            (map sample
                 (multinomial
                  (list (terminal "cooks")
                        (terminal "works")
                        (terminal "makes"))
                  (list θ1^V θ2^V θ3^V)))))

(define A (lambda ()
            (map sample
                 (multinomial
                  (list (terminal "diligently"))
                  (list θ1^A)))))
```

```
(define AP (lambda ()
             (map sample
                  (multinomial
                   (list (list A))
                   (list θ1^AP)))))

(define NP (lambda ()
             (map sample
                  (multinomial
                   (list (list D N))
                   (list θ1^NP)))))

(define VP (lambda ()
             (map sample
                  (multinomial
                   (list (list V AP)
                         (list V NP))
                   (list θ1^VP θ2^VP)))))

(define S (lambda ()
            (map sample
                 (multinomial
                  (list (list NP VP))
                  (list θ1^S)))))
```

Figure 4: Church code defining the generative process for a probabilistic context-free grammar.

Figure 5 zooms in on just the `VP` procedure from the above code listing. `VP` makes use of the following procedures in its definition.

- `map` is a procedure that takes a two arguments: another procedure, and a list. It returns the list that results from applying the other procedure to each element of the input list.

- `sample` is a procedure which takes a thunk and draws a sample from it.
- `list` is a procedure which takes any number of arguments and returns a list containing all of those arguments.

Working from the inside out, we see that the `VP` procedure is built around a multinomial distribution. This multinomial is over the two possible RHSs of `VP` in our grammar, each RHS is represented as a list. This multinomial distribution is given a list containing these two RHSs as well as list of probabilities, (`list` $\theta_1^{\texttt{VP}}$ $\theta_2^{\texttt{VP}}$), specifying the weights for the RHSs. When called, the `VP` procedure will first call this multinomial, which will sample and return one of the two possible RHSs as a list. It will then call the `map` procedure. `map` will apply `sample` to each element of the list. Since the list consists of names of other procedures, calling `sample` on each one will implement the basic recursion of the PCFG.

```
(define VP (lambda ()
            (map sample
                 (multinomial
                  (list (list V AP)
                        (list V NP))
                  (list θ1^VP  θ2^VP)))))
```

Figure 5: The procedure `VP`

The other procedure definitions work similarly. The only remaining detail is that the procedure `terminal` does not recurse; it simply returns its arguments.

2.1.1 PCFG Distributions

In this section we will consider in more detail the distribution over computations and expressions defined by a PCFG. As mentioned earlier, a parse tree is a tree representing the trace of the computation of some expression e from some nonterminal category `A`. We will call a parse tree *complete* if its leaves are all terminals; that is, if it represents a complete computation of procedure `A`. A tree *fragment* is a tree whose leaves may be a mixture of nonterminals and terminals. Given a tree, the procedure `yield` returns the leaves of a tree (fragment) as a list. The procedure `root` returns the procedure (name) at the root of the tree. When it is clear from context we will sometimes abuse the meaning of the `root` function to also return the rule at the top of the tree (that is, the depth-one tree that corresponds to the root node plus its children).

We will say that a nonterminal `A` derives some expression, represented as a list of terminals $\vec{w} \in W^*$, if there is a complete tree t such that `(root` t`) = A` and `(yield` t`) =` $\vec{w}$. The *language* associated with nonterminal `A` is the set of expressions which can be computed by that nonterminal.

We define a *corpus* $\boldsymbol{E}$ of expressions of size $N_{\boldsymbol{E}}$ with respect to a grammar to be the result of executing the procedure `S` $N_{\boldsymbol{E}}$ times: `(repeat` $N_{\boldsymbol{E}}$ `S)` where the procedure `(repeat num proc)` returns the list resulting from applying the procedure `proc` `num` times.

Formally, a (multinomial) PCFG, $\langle G, \Theta \rangle$, is a CFG G together with a set of vectors $\Theta = \{\vec{\theta}^{\mathtt{A}}\}$. Each vector $\vec{\theta}^{\mathtt{A}}$ represents the parameters of a multinomial distribution over the set of rules that share `A` on their left-hand sides. We write $\theta^{\mathtt{A}}_r$ or θ_r to mean the component of vector $\vec{\theta}^{\mathtt{A}}$ associated with rule r (that is with the specific RHS of rule r). Θ satisfies:

$$\sum \vec{\theta}^{A} = 1$$

As discussed above, the probability distribution of a random procedure is defined by taking the product of the probabilities of (the return values of) each elementary random procedure evaluated in its body. In the case of a PCFG the only randomness is in the multinomial distributions associated with choosing a RHS for each nonterminal. Thus the probability of a particular parse tree t is given by:

$$P(t|G) = \prod_{r \in t} \theta_r \tag{1}$$

The probability of a particular expression $\vec{w}$ is computed by marginalizing over all derivation trees which share that expression as their yield.

$$P(\vec{w}|G) = \sum_{t \ | \ (\mathtt{yield\ t}) = \vec{w}} P(t|G)$$

Given an expression $\vec{w}$ and a rule r we define the *inside probability* of $\vec{w}$ given r as:

$$P(\vec{w}|r, G) = \sum_{t \ | \ (\mathtt{yield\ t}) = \vec{w} \ \wedge \ (\mathtt{root\ t}) = r} P(t|G) \tag{2}$$

The inside probability of a string given a rule is the probability that that string is the yield of a complete tree whose topmost subtree corresponds to that rule.[10]

An important feature of PCFGs is that they make two strong conditional independence assumptions. First, all decisions about expanding a parse tree are local to the procedure. They cannot make reference to any other information in the parse tree. Second, sentences themselves are generated independently of one another; there is no notion of history in a PCFG. These conditional independence assumptions result in the existence of efficient algorithms for PCFG parsing and training, but they also make PCFGs inadequate as models of natural language structure. The models we develop below can be seen as a collection of ways of relaxing the conditional independence assumptions of PCFGs.

Let $\boldsymbol{E} = \{e^{(i)}\}$ be a corpus of expressions, and let $\boldsymbol{P} = \{p^{(i)}\}$ be a set of parse trees specifying the exact way each expression was generated. Let $\boldsymbol{X} = \{\vec{x}^{\mathtt{A}}\}$ be the set of count vectors for each nonterminal multinomial in the grammar. The procedure `counts` takes a set of parse trees and returns the corresponding counts: `(counts` $\boldsymbol{P}$ `)` $= \boldsymbol{X}$. We will occasionally abuse the meaning of this procedure and also use it to return the counts associated with some particular parse tree: `(counts` $p^{(i)}$ `)` $= \boldsymbol{x}_{p^{(i)}}$.

The conditional independence assumptions on PCFGs mean that we can compute the probability of a corpus of expressions and parse trees by taking the product of rule choices in the parse trees in any order we like. Moreover, the counts of rule uses are the *sufficient* statistics for the multinomials for each nonterminal. This means that the probability of a set of parses can be calculated purely from the count information.

Thus the probability of a corpus of expressions $\boldsymbol{E}$ and corresponding parse trees $\boldsymbol{P}$ can be given in terms of the corresponding count vectors `(counts` $\boldsymbol{P}$ `)` $= \boldsymbol{X}$:

$$\begin{aligned} P(\boldsymbol{E}, \boldsymbol{P}|G) &= \mathbf{pcfg}(\boldsymbol{X}; G) \\ &= \prod_{\mathtt{A} \in V} \prod_{r \in R^{\mathtt{A}}} [\theta^{\mathtt{A}}_r]^{x^{\mathtt{A}}_r} \end{aligned} \tag{3}$$

Where $x^{\mathtt{A}}_r$ is the number of times that rule r with LHS `A` was used in the corpus.

[10]Note that we have defined inside probabilities with respect to rules. Usually they are defined with respect to nonterminal categories. The inside probability of a string $\vec{w}$ with respect to a nonterminal `A` is computed by additionally marginalizing over rules that share `A` on their LHS:
$$P(\vec{w}|\mathtt{A}, G) = \sum_{r \ | \ (\mathtt{lhs}\ r) = \mathtt{A}} \left[\sum_{t \ | \ (\mathtt{yield\ t}) = \vec{w} \ \wedge \ (\mathtt{root\ t}) = r} P(t|G) \right].$$

3 Stochastic Memoization

In this section we develop the notion of stochastic memoization which we will use to formalize our proposal about reuse in language structure. *Memoization* refers to the technique of storing the results of computation for later reuse. In situations where identical sub-computations happen repeatedly in the context of a larger computation, the technique can significantly reduce the cost of executing a program.

```
(define fib (lambda (n)
              (case n
                (0 0)
                (1 1)
                (else
                 (+ (fib (- n 1)) (fib (- n 2)))))))
```

Figure 6: Procedure to compute the nth Fibonacci number.

This idea can be best illustrated with an example. Figure 6 shows Scheme code to compute the nth Fibonacci number. This code says: if n is 0 or 1, then return 0 or 1, respectively; otherwise we can compute the nth Fibonacci number by first computing the $(n-1)$th and $(n-2)$th Fibonacci numbers and adding them. The tree tracing the computation of `(fib 6)` is shown in Figure 7.

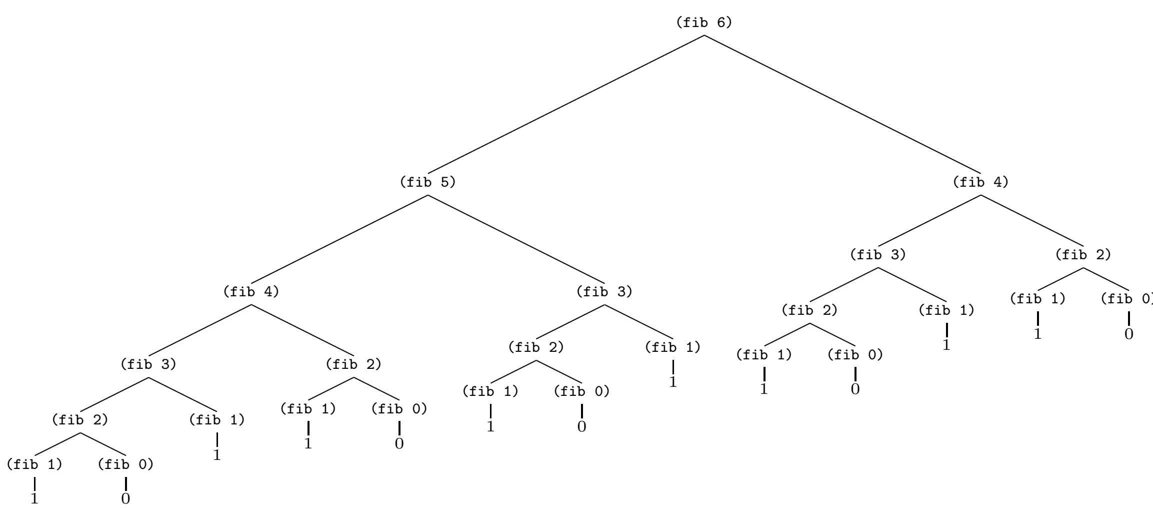

Figure 7: Computation of `(fib 6)` without memoization.

What can be seen from this example is that `fib` repeats a lot of computation. For instance, in this tree `(fib 2)` is evaluated 5 times, and `(fib 4)` is evaluated twice. By memoizing the result of each of these evaluations the first time they are evaluated, and then reusing those results, we can save a lot of work. This is represented in Figure 8. In the case of the `fib` procedure in particular, we can turn a computation that takes an exponential number of steps into a computation that takes only a linear number of steps by memoizing.

Memoization has been applied widely in the design of algorithms, especially *dynamic programming* algorithms, which figure prominently in linguistic applications such as chart parsing (see, e.g., [23, 27]). It has also played a prominent role in the implementation of functional programming languages.[11]

To memoize a deterministic procedure, we need to maintain a table of input–output pairs, called a *memotable* When the procedure is applied to an argument, we intercept the call to the procedure and first consult the memotable to see if the result has already been computed. If it has, we return the previously computed value. If the procedure has not been applied to these arguments before, then we compute the value, save it on the memotable, and return it. In programming languages with first-class procedures, such as Scheme, memoization can easily be added as a *higher-order* procedure, that is, a procedure which takes another procedure as an argument. We will assume a higher-order procedure `mem` which takes as an argument a procedure and returns a memoized version of it.

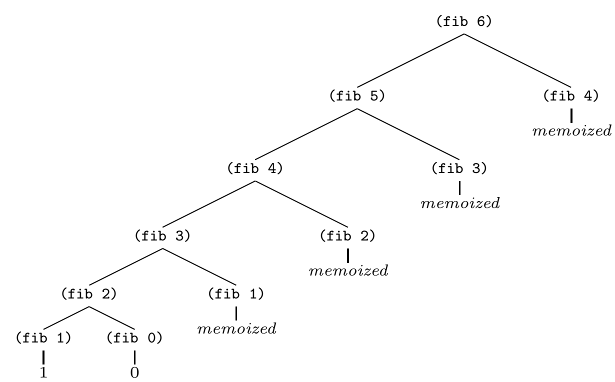

Figure 8: Computation of `((mem fib) 6)`.

In a stochastic setting, a procedure applied to some inputs is not guaranteed to evaluate to the same value every time. If we wrap such a random procedure

[11] See [31, 20] for a discussion of the relationship between memoization in functional programming languages and CFG parsing algorithms.

in a deterministic memoizer, then it will sample a value the first time it is applied to some arguments, but forever after, it will return the same value by virtue of memoization. It is natural to consider making the notion of memoization itself stochastic, so that sometimes the memoizer returns a value computed earlier, and sometimes it computes a fresh value.

A stochastic memoizer wraps a stochastic procedure in another distribution, called the *memoization distribution*, which tells us when to reuse one of the previously computed values, and when to compute a fresh value from the underlying procedure. To accomplish this, we generalize the notion of a memotable so that it stores a *distribution* for each procedure–plus–arguments combination [17]. In the next section we describe these distributions.

3.1 Memoization Distributions

Deterministic memoization is important precisely because much of the work that we do in any particular computation can be reused. In SFP we work with random procedures because we want to express uncertainty in computation. By analogy, a stochastic memoizer should capture uncertainty in the reuse of computation. A sensible memoization distribution should be sensitive to the number of times a particular value was computed in the past, favoring those values which often proved useful. In this section, following [21, 17], we develop a memoization distribution based on the *Chinese restaurant process* (CRP).

3.1.1 The Chinese Restaurant Process

The Chinese restaurant process is distribution from non-parametric Bayesian statistics. The term *non-parametric* refers to statistical models whose size or complexity can grow with the data, rather than being specified in advance. The CRP is usually described as a sequential sampling scheme using the metaphor of a restaurant.

We imagine a restaurant with an infinite number of tables. The first customer enters the restaurant and sits at the first unoccupied table. The $(N+1)$th customer enters the restaurant and sits at either an already occupied table or a new, unoccupied table, according to the following distribution.

$$\tau^{(N+1)}|\tau^{(1)}, ..., \tau^{(N)}, \alpha \sim \sum_{i=1}^{K} \frac{y_i}{N+\alpha}\delta_{\tau_i} + \frac{\alpha}{N+\alpha}\delta_{\tau_{K+1}}$$

N is the total number of customers in the restaurant. K is the total number of occupied tables, indexed by $1 \geq i \geq K$. $\tau^{(j)}$ refers to the table chosen by the jth customer. τ_i refers to ith occupied table in the restaurant. y_i is the

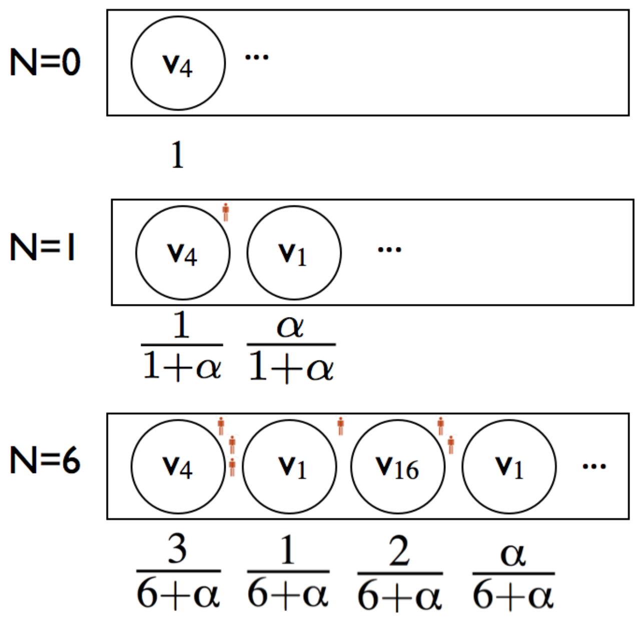

Figure 9: A series of possible distributions generated by the Chinese restaurant process. Shown is the distribution over the next customer after N customers have already been seated. The values $\mathtt{v}_i$ have been drawn from some associated *base distribution* μ.

number of customers seated at table τ_i; δ_τ is the δ-distribution which puts all of its mass on table τ. $\alpha \geq 0$ is the *concentration parameter* of the model.

In other words, customers sit at an already-occupied table with probability proportional to the number of individuals at that table, or at a new table with probability controlled by the parameter α. This is illustrated in Figure 9.

Each table has a *dish* associated with it. Each dish `v` is a label on the table which is shared by all the customers at that table. When a customer sits at a new table, τ_i, a dish is sampled from another distribution, μ, and placed on that table. This distribution, μ, is called the *base distribution* of the Chinese restaurant process, and is a parameter of the model. From then on, all customers who are seated at table τ_i share this dish, $\mathtt{v}_{\tau_i}$.

To use a CRP as a memoization distribution we let our memotable be a set of restaurants—one for each combination of a procedure with its arguments. For example, consider the procedure in Figure 2 again. This procedure can take a variety of different kinds of arguments: `(noisy-or 1 1)`, `(noisy-or 0 1)`, `(noisy-or 0 0 1)`, `(noisy-or 0 0 0 1)`, etc. We associate a restaurant with each of these combinations of arguments as shown in Figure 10. We let customers represent particular instances in which a procedure is applied, and we let the dishes labeling each table represent the values that result from those procedure applications. The base distribution which generates dishes corresponds to the underlying procedure which we have memoized.

When we seat a customer at an existing table, it corresponds to retrieving a value from our memotable. Every customer seated at an existing table always returns the dish placed at that table when it was created. When we seat a customer at a new table it corresponds to computing a fresh value from our memoized random function and storing it as the dish at the new table.

Another way of understanding the CRP is to think of it as defining a distribution over ways of partitioning N items (customers) into K partitions (tables), for all possible N and K.

The probability of a particular partition of N customers over K tables is the product of the probabilities of the N choices made in seating those customers. It can easily be confirmed that the order in which elements are added to the partition components does not affect the probability of the final partition (i.e. the terms of the product can be rearranged in any order). Thus the distribution defined by a CRP is *exchangeable*.

A sequence of random variables is exchangeable if it has the same joint distribution under all permutations. Intuitively, exchangeability says that the order in which we observed some data will not make a difference to our inferences about it. Exchangeability is an important property in Bayesian statistics, and our inference algorithms below will rely on it crucially. It is also a desirable property in cognitive models.

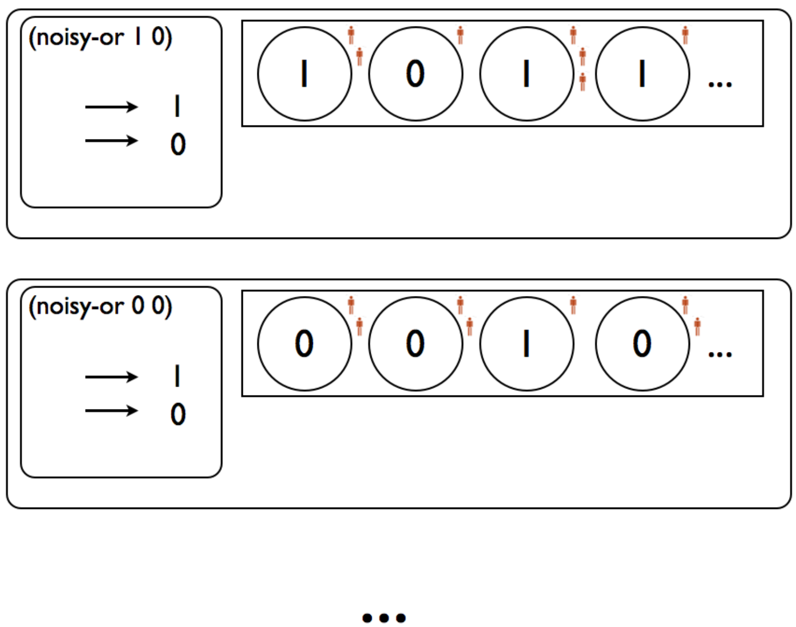

Figure 10: Stochastic memoization. To stochastically memoize a procedure like `noisy-or`, that is to construct `(CRPmem` α `noisy-or)`, we associate a CRP with each possible combination of input arguments.

The probability of a particular CRP partition can also be written down in closed form as follows.

$$P(\vec{y}) = \frac{\alpha^K \Gamma[\alpha] \prod_{j=0}^{K} \Gamma[y_j]}{\Gamma[\alpha + \sum_{j=0}^{K} y_j]} \tag{4}$$

Where $\overrightarrow{y}$ is the vector of counts of customers at each table and $\Gamma(\cdot)$ is the gamma function, a continuous generalization of the factorial function. This shows that for a CRP the vector of counts is sufficient.

As a distribution, the CRP has a number of useful properties. In particular, it implements a simplicity bias. It assigns a higher probability to partitions which: *a.*) have fewer customers *b.*) have fewer tables *c.*) for a fixed number of customers N, assign them to the smallest number of tables. Thus the CRP favors simple restaurants and implements a rich-get-richer scheme. Tables with more customers have higher probability of being chosen by later customers. These properties mean that, all else being equal, when we use the CRP as a stochastic memoizer we favor reuse of previously computed values.

3.1.2 Pitman-Yor processes

In the models we discuss below, the memoization distribution used is actually a generalization of the CRP known as the Pitman-Yor process (PYP). The Pitman-Yor process is identical to the CRP except for having an extra parameter, a, which introduces a dependency between the probability of sitting at a new table and the number of tables already occupied in the restaurant.

The process is defined as follows. The first customer enters the restaurant and sits at the first table. The $(N+1)$th customer enters the restaurant and sits at either an already occupied table or a new one, according to the following distribution.

$$\tau^{(N+1)} | \tau^{(1)}, ..., \tau^{(N)}, a, b \sim \sum_{i=1}^{K} \frac{y_i - a}{N + b} \delta_{\tau_i} + \frac{Ka + b}{N + b} \delta_{\tau_{K+1}}$$

Here all variables are the same as in the CRP, except for a and b. $b \geq 0$ corresponds to the CRP α parameter. $0 \leq a \leq 1$ is a new *discount* parameter which moves a fraction of a unit of probability mass from each occupied table to the new table. When it is 1, every customer will sit at their own table. When it is 0 the distribution becomes the single-parameter CRP [33]. The a parameter can be thought of as controlling the *productivity* of a restaurant: how much sitting at a new table depends on how many tables already exist. On average, a will be the limiting proportion of tables in the restaurant which have only a single customer. The b parameter controls the rate of growth of new tables in relation to the total number of customers N as before [36].

Like the CRP, the sequential sampling scheme outlined above generates a distribution over partitions for unbounded numbers of objects. Given some vector of table counts $\vec{y}$, A closed-form expression for this probability can be given as follows. First, define the following generalization of the factorial function, which multiples m integers in increments of size a starting at x.

$$[x]_{m,s} = \begin{cases} 1 & \text{for } m = 0 \\ x(x+s)...(x+(m-1)s) & \text{for } m > 0 \end{cases} \tag{5}$$

Note that $[1]_{m,1} = m!$.

The probability of the partition given by the count vector, $\vec{y}$, is defined by:

$$P(\vec{y}|a,b) = \frac{[b+a]_{N-1,a}}{[b+1]_{K-1,1}} \prod_{i=1}^{K} ([1-a]_{i-1,1})^{y_i} \tag{6}$$

It is easy to confirm that in the special case of $a = 0$ and $b > 0$, this reduces to the closed form for CRP by noting that $[1]_{m,1} = m! = \Gamma[m+1]$. In what follows, we will assume that we have a higher-order function `PYmem` which takes three arguments `a`, `b`, and `proc` and returns the PYP-memoized version of `proc`.

3.1.3 Multinomial-Dirichlet Distributions

In this section we define what is known as the Polya urn representation of the multinomial-Dirichlet distribution. This section and the following section on multinomial-Dirichlet PCFGs are important for understanding the details of the fragment grammar model, but can be skipped on first reading.

Imagine a multinomial distribution over K elements with parameters specified by parameter vector $\vec{\theta}$. Rather than specifying $\vec{\theta}$ as a parameter, one can define a hierarchical model where $\vec{\theta}$ is itself drawn from a prior distribution. The most common prior on multinomial parameters is the Dirichlet distribution.[12]

The combination of a multinomial together with a Dirichlet prior can be represented by a sequential sampling scheme which is the finite analog of the Chinese restaurant process. That is, we can think of the combination multinomial-Dirichlet distribution as a distribution assigning probabilities to

[12] The Dirichlet is commonly used as the prior for the multinomial because it is *conjugate* to the multinomial. This means that the posterior of a multinomial-Dirichlet distribution is another Dirichlet distribution [12].

partitions of N objects amongst K bins *for fixed* K. This representation of the MDD can be defined via a sequential sampling construction which is very similar to that of the CRP. For a discussion of this mysterious fact, please see Appendix A.

Suppose that we have a finite set of K values. We define the following sequential process. We sample our first observation $\mathtt{v}^{(1)}$ according to the following equation.

$$\mathtt{v}^{(1)}|\pi_1, ..., \pi_K \sim \frac{\pi_1}{\sum_{i=1}^{K} \pi_i}\delta_{\mathtt{v}_1} + ... + \frac{\pi_K}{\sum_{i=1}^{K} \pi_K}\delta_{\mathtt{v}_K}$$

Where the πs are *pseudocounts*, which can be thought of as imaginary prior observations of each of the K possible outcomes. After N observations have been sampled, the $N + 1$th observation is sampled as follows:

$$\mathtt{v}^{(N+1)}|\mathtt{v}^{(1)}, ..., \mathtt{v}^{(N)}, \pi_1, ..., \pi_K \sim \frac{\pi_1 + x_1}{\sum_{i=1}^{K}[\pi_i + x_i]}\delta_{\mathtt{v}_1} + ... + \frac{\pi_K + x_K}{\sum_{i=1}^{K}[\pi_i + x_i]}\delta_{\mathtt{v}_K}$$

x_i is the number of draws of value i, δ_i is a δ-distribution, in other words a distribution which puts probability 1 on a single value—in this case partition i. $\vec{\pi}$ is a vector of length K of *pseudocounts* for each of the K values. The pseudocounts can be thought of as “imaginary” counts of each the values we have “observed” prior to using the distribution. They give us a prior weight for each value.

Note that unlike the CRP and PYP, we can draw values $\mathtt{v}$ directly from the multinomial-Dirichlet distribution. This is a consequence of the fact that our set of values is finite. Under the multinomial-Dirichlet distribution, each time we draw an observation we become more likely to draw it again in the future. This shows that each draw from a multinomial-Dirichlet distribution is dependent on the history of prior draws, and implements a rich-get-richer dynamic much like the CRP and PYP discussed above.

The distribution over the entire partition is once again given by the product of the probabilities of each choice made in its construction. This distribution can be easily confirmed to be exchangeable. The probability of the partition given by the count vector, $\vec{x}$, is given by:

$$P(\vec{x}|\vec{\pi}) = \frac{\prod_{i=1}^{K}\Gamma(\pi_i + x_i)}{\Gamma(\sum_{i=1}^{K}\pi_i + x_i)}\frac{\Gamma(\sum_{i=1}^{K}\pi_i)}{\prod_{i=1}^{K}\Gamma(\pi_i)} \tag{7}$$

For further discussion of these distributions please see Appendix A.

4 Multinomial Dirichlet PCFGs

In this section we define Multinomial Dirichlet PCFGs (MD-PCFGs), which are PCFGs in which Dirichlet priors have been put on the rule weights. Both the adaptor grammar model and the fragment grammar model discussed below use MD-PCFGs as their base distributions. For the sake of clarity, however, this detail has been suppressed in later discussions. This section includes the details of MD-PCFGs for completeness, but can be skipped upon first reading.

For simple PCFGs the set of weight vectors Θ is specified in advance as a parameter of the model. It is possible to define an alternate model, however, where these vectors are drawn from a Dirichlet prior [22, 26]. We call the resulting model a *multinomial-Dirichlet probabilistic context-free grammar* (MD-PCFG).[13] A MD-PCFG $\mathcal{G} = \langle G, \Pi \rangle$ is a context-free grammar together with a set $\Pi = \{\vec{\pi}^{\mathtt{A}}\}$ of vectors of pseudocounts for Dirichlet distributions associated with each nonterminal `A`.

As in the case of the multinomial distribution, the set of counts vectors $\boldsymbol{X} = \{\vec{x}^{\mathtt{A}}\}$ is sufficient for the multinomial-Dirichlet distribution (see section 3.1.3 above). Thus the probability of a corpus $\boldsymbol{E}$ together with parses for each expression $\boldsymbol{P}$ under an MD-PCFG $\mathcal{G}$ can be given in closed form as follows:

$$
\begin{aligned}
P(\boldsymbol{E}, \boldsymbol{P} | \mathcal{G}) &= \mathbf{mdpcfg}(\boldsymbol{X}; \mathcal{G}) \\
&= \prod_{\mathtt{A} \in V} \left[\frac{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})}{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})} \frac{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}})}{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}})} \right]
\end{aligned}
\tag{8}
$$

We can represent a hypothetical state of an MD-PCFG as in Figure 11.

5 Adaptor Grammars

A Pitman-Yor adaptor grammar (PYAG), first presented in [21], is a CFG where each nonterminal procedure has been stochastically memoized. This means that PYAGs memoize the process of deriving an expression itself. This is shown in the code listing in Figure 12.

As can be seen from the code, each nonterminal procedure is now stochastically memoized using `PYmem`. Each time one of these procedures is called, the call will be intercepted and either *a.*) a previously computed expression will be

[13] As discussed in Appendix A, if the draws of each of the $\{\vec{\theta}^{\mathtt{A}}\}$ are unobserved, then we can equivalently express these distributions with hierarchical de Finetti representations or as a sequential sampling scheme.

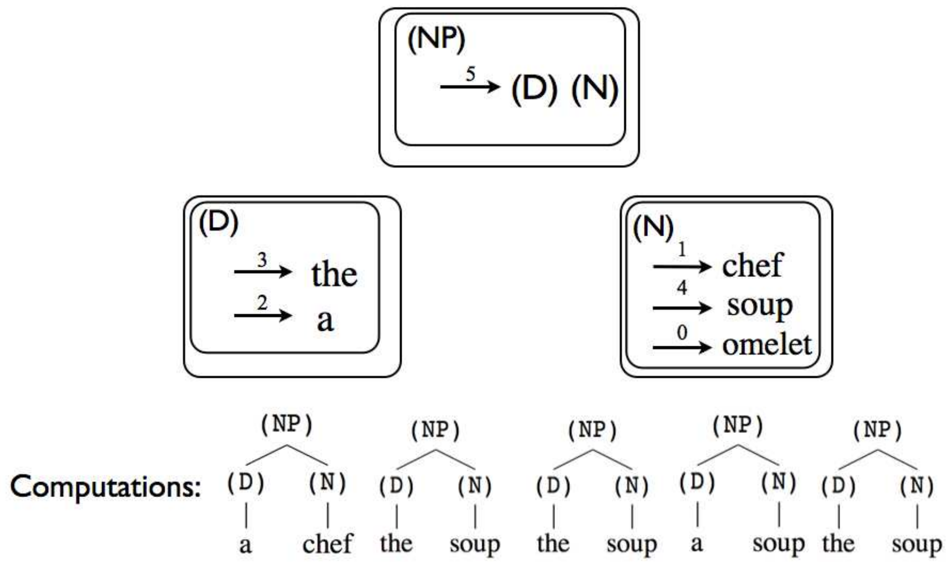

Figure 11: Representation of a possible state of an multinomial-Dirichlet PCFG after having computed the five expressions shown at the bottom. The numbers over the rule arrows represent the counts of each rule in the set of parses.

```
(define D (PYmem a^D b^D
               (lambda ()
                 (map sample
                      (multinomial
                       (list (terminal "the")
                             (terminal "a"))
                       (list θ_1^D θ_2^D))))))

(define N (PYmem a^N b^N
               (lambda ()
                 (map sample
                      (multinomial
                       (list (terminal "chef")
                             (terminal "soup")
                             (terminal "omelet"))
                       (list θ_1^N θ_2^N θ_3^N))))))

(define V (PYmem a^V b^V
               (lambda ()
                 (map sample
                      (multinomial
                       (list (terminal "cooks")
                             (terminal "works")
                             (terminal "makes"))
                       (list θ_1^V θ_2^V θ_3^V))))))

(define A (PYmem a^A b^A
               (lambda ()
                 (map sample
                      (multinomial
                       (list (terminal "diligently"))
                       (list θ_1^A))))))

(define AP (PYmem a^AP b^AP
               (lambda ()
                 (map sample
                      (multinomial
                       (list (list A))
                       (list θ_1^AP))))))

(define NP (PYmem a^NP b^NP
               (lambda ()
                 (map sample
                      (multinomial
                       (list (list D N))
                       (list θ_1^NP))))))

(define VP (PYmem a^VP b^VP
               (lambda ()
                 (map sample
                      (multinomial
                       (list (list V AP)
                             (list V NP))
                       (list θ_1^VP θ_2^VP))))))

(define S (PYmem a^S b^S
              (lambda ()
                (map sample
                     (multinomial
                      (list (list NP VP))
                      (list θ_1^S))))))
```

Figure 12: Church code for a PYAG.

returned, or *b*.) a new expression will be computed, stored in the memotable, and then returned. [14]

5.1 A Representation for Adaptor Grammar States

The code above precisely describes the adaptor grammar generative model. However, in order to understand the properties of the model more deeply, it is useful to discuss the representation of the state of an adaptor grammar at a given point in time, after having sampled some number of expressions.

Figure 13 shows a possible adaptor grammar state after five calls to the `NP` procedure. The parse trees for each call are shown at the bottom of the

[14] It is not yet clear under what conditions adaptor grammars built on recursive CFGs are well-defined. This is also true for the models discussed later in the paper. This is a topic of ongoing research. In this technical report we restrict ourselves to non-recursive grammars which are well-defined by merit of being instances of hierarchical Dirichlet processes.

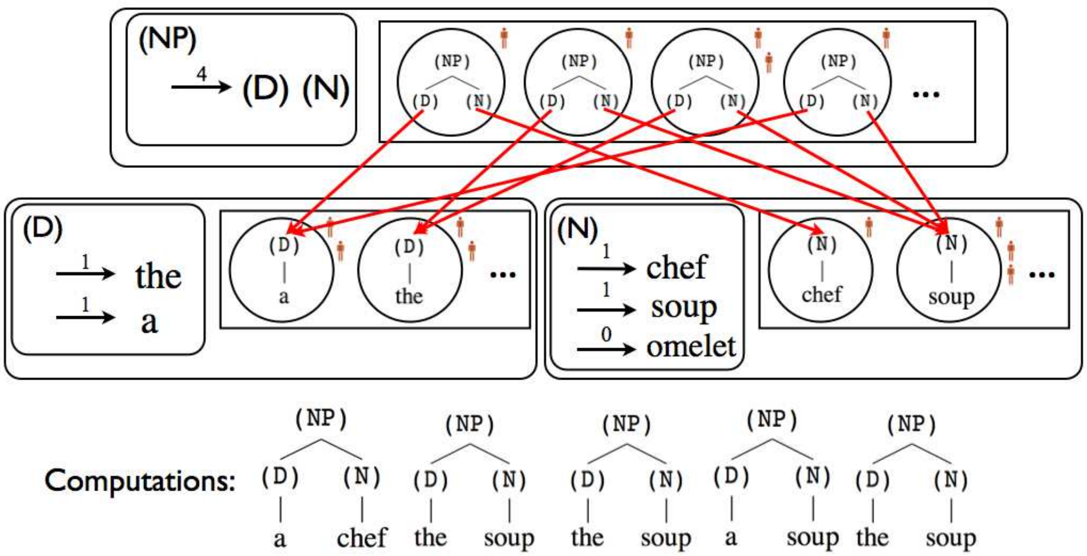

Figure 13: Representation of a possible state of an adaptor grammar after having computed the five expressions shown at the bottom.

figure. The figure has been drawn to show the trace of the computations for each call. This representation of the state actually contains more information than is really stored in the memotables for each nonterminal procedure. What is actually stored on each table in a restaurant is just the computed expression that resulted from the call to the base distribution. However, for clarity we show (in red) the traces of computation that resulted from that call.

The five calls to `NP` resulted in the creation of four tables, and the reuse of one computed expression at the third table. Applications of `NP` resulted in the hierarchical application of the `D` and `N` procedures. Red arrows show which particular calls to `D` and `N` resulted in the subexpressions associated with the tables in the `NP` restaurant.

A few important things to note about Figure 13.

- The expressions stored at the second and third tables are identical: ⟨the soup⟩. Draws from the base distribution for a memoized procedure are independent, therefore it is possible to draw the same table labels multiple times.
- The number of customers associated with a table only increases when the table is reused in a computation. In particular, the third table, corresponding to the expression ⟨the soup⟩ gets reused at the level of `NP`. This does *not* lead to the tables for ⟨the⟩ and ⟨soup⟩ getting incremented. The subexpression tables were only incremented when the table was first created. It was only then that the procedures `D` and `N` were evaluated.
- The numbers over the arrows in the underlying procedures represent the counts associated with the right-hand sides of the underlying CFG rules that have been sampled while generating a table label. Notice that these counts correspond to the number of tables which were built using that rule, *not* to the number of times each table was reused. In other words, the distribution over underlying rules only gets updated when new tables are created.

  Adaptor grammars inherit this property from CRPs. An important debate within linguistics has been whether the probability of a rule should be estimated based on its *token* count—that is, the count of the number of times the rule occurs in a corpus—or its *type* count—that is, the count of the number of different forms it appears with in the corpus [2]. It has been argued that the ability of CRPs and PYPs to interpolate between type and token counts in this way is an important advantage for linguistic applications [15, 36].[15]

[15] Moreover, in the limit, CRP/Pitman-Yor distributions show a power-law distribution over frequencies of table labels [33], consistent with Zipf's well-known observations about word token

When an adaptor grammar reuses a table, it is reusing previous computation. Thus, according to our discussion in the introduction, the labels on adaptor grammar tables correspond to lexical items.

6 Lexcial Items as Procedures and Two-stage Generative Models

We mentioned two novel ideas expounded in this report. First is the notion of lexical items as distributions—or, equivalently in the context of stochastic functional programming—as procedures of no arguments (thunks). The second is two-stage interpretation of our basic generative process. First, we build a lexical item—deciding what parts of the current computation to store for later reuse—and then we use that lexical item procedure to sample an expression.

We will illustrate these ideas in the following section in terms of adaptor grammars. In the case of adaptor grammars this generalization does not change the behavior of the model. In other words, for adaptor grammars these ideas are meaningless. However, with these changes in place we will be able to define fragment grammars with just a few small changes to our Church code.

6.1 Adaptor Grammars as Two-stage Models

Consider a PYAG which has generated the single expression $\langle$ a chef $\rangle$, as in Figure 14.

The procedure `NP` in a PYAG or PCFG defines some distribution over noun-phrase expressions. In the case of the grammar in our running example, the support of this distribution—the set of items that have positive probability—is finite. An example of such a distribution is shown on the left hand side of Figure 15.

When we sample from our PYAG we create a table associated with the expression $\langle$ a chef $\rangle$. Up till now, we have thought of using this expression itself as the label on the table we created. However, we will now pursue the alternative outlined in the introduction and think of this lexical item as being a distribution.

What kind of distribution is labeling this table? In this case, it is the δ-distribution (point distribution) which concentrates all of its mass on a single expression. This is shown on the right hand side of Figure 15. In Church, distributions are represented as procedures of no arguments. A δ-distribution

frequencies [38].

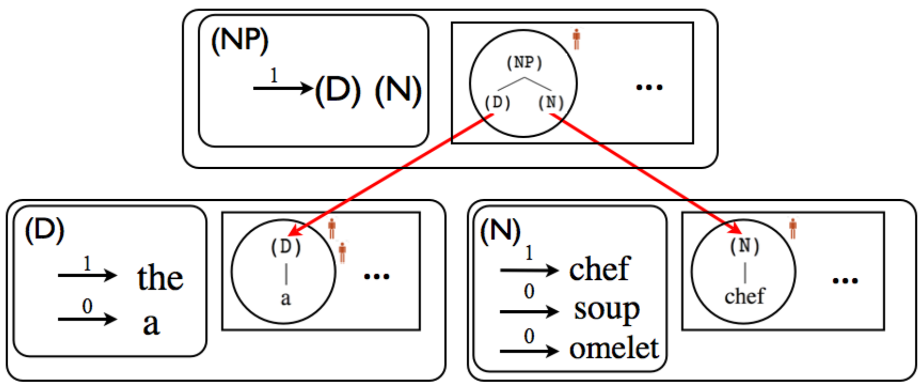

Figure 14: State of a PYAG after having generated the expression ⟨ a chef ⟩.

on a single expression is a procedure of no arguments which always returns the same value, (also known as a deterministic thunk): `(lambda ()` $\langle$ a chef $\rangle$ `)`.

Figure 15: Creating a table in a PYAG can be thought of as concentrating a distribution around a single expression. In this case, creating a table corresponding to the expression $\langle$ a chef $\rangle$ can be thought of as concentrating the `NP` distribution into the δ-distribution: `(lambda ()` $\langle$ a chef $\rangle$ `)`.

The next draw from `NP` is a draw from the mixture distribution over this δ-distribution and the underlying base distribution. This is shown in Figure 16.

The preceding discussion suggests an alternate, two-stage view of the adaptor grammar generative process. To sample from the procedure `A` *1.*) draw a **lexical item** from the memoizer associated with `A`, which in this case is a Pitman-Yor process, and *2.*) draw an expression from that lexical item. .

To define the two-stage generative model for a PYAG we will first need the helper procedures shown in Figure 17. The procedure `sample-delta-distribution` implements the basic recursion that builds lexical items in the PYAG. It samples an expression, wraps it in a procedure of no arguments, and then returns it. The procedure `expand-lexical-item` simply `maps` `sample-delta-distribution` across the right-hand-side of a PCFG rule.

With these two procedures defined, we can now define the entire two-stage version of the PYAG generative process. This is shown in the code listing in Figure 18

When defining a procedure `A`, we first construct a random procedure which defines a distribution over lexical items. This procedure is encapsulated inside the `A` procedure. The `A` procedure itself first evaluates this closed-over distribution to draw a lexical item, and then samples the lexical item it drew to produce an expression.

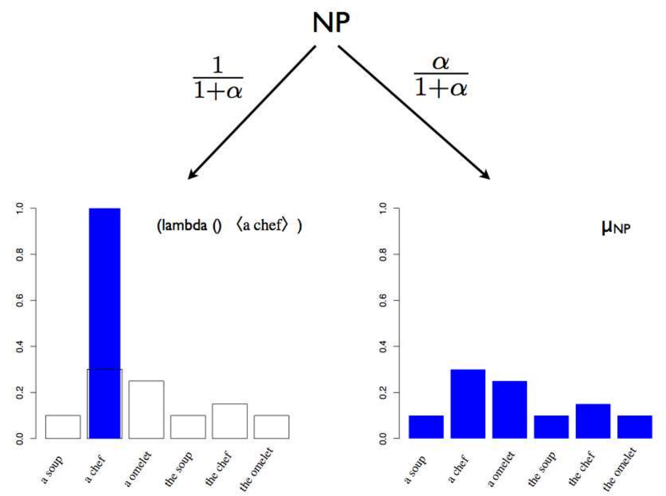

Figure 16: After having generated the expression ⟨ a chef ⟩ the next draw from the procedure `NP` is a draw from the mixture over the δ-distribution and the base distribution of `NP`, $\mu_{\texttt{NP}}$.

```
(define (sample-delta-distribution proc)
  (let ((value (sample proc)))
    (lambda () value)))

(define (expand-lexical-item right-hand-side)
  (map sample-delta-distribution right-hand-side))
```

Figure 17: Helper procedures for generating lexical items in an adaptor grammar.

The two-stage perspective on adaptor grammars also clarifies the structure of an adaptor grammar parse tree, which must contain information not present in a (MD-)PCFG parse. While the parses for (MD-)PCFGs simply specify the series of procedure calls which lead to an expression, an adaptor grammar parse tree must also specify, for each procedure call, which particular lexical item was returned and used on that call.

Formally, an adaptor grammar, $\mathcal{A} = \langle G, \{\vec{\pi}^{\mathtt{A}}\}, \{\langle a^A, b^A \rangle\}\rangle$, is a distribution on distributions of expressions. G is a context free grammar. $\{\vec{\pi}^{\mathtt{A}}\}$ is a set of hyperparameter vectors for multinomial-Dirichlet distributions for each nonterminal `A` and $\{\langle a^A, b^A \rangle\}$ is a set of hyperparameters for Pitman-Yor processes for each nonterminal `A`. Let $\boldsymbol{X} = \{\vec{x}^{\mathtt{A}}\}$ be the set of count vectors for underlying CFG rule uses as before, and let $\boldsymbol{Y} = \{\vec{y}^{\mathtt{A}}\}$ be the a set of count vectors for lexical item uses in generating $\boldsymbol{E}$. We will use the shorthand $\boldsymbol{A} = \langle \boldsymbol{X}, \boldsymbol{Y} \rangle$ for the representation of an adaptor grammar state in terms of sufficient statistics (counts) after having generated $\boldsymbol{E}$. The joint probability of the corpus together with parses $\boldsymbol{P}$ for that corpus is given by:

$$
\begin{aligned}
P(\boldsymbol{E}, \boldsymbol{P} | \mathcal{A}) &= \mathbf{pyag}(\boldsymbol{A}; \mathcal{A}) \qquad (9) \\
&= \prod_{\mathtt{A} \in V} \left[\frac{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})}{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})} \frac{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}})}{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}})} \frac{[b^{\mathtt{A}} + a^{\mathtt{A}}]_{N^{\mathtt{A}}-1, a^{\mathtt{A}}}}{[b^{\mathtt{A}} + 1]_{K^{\mathtt{A}}-1,1}} \prod_{i=1}^{K^{\mathtt{A}}} ([1 - a^{\mathtt{A}}]_{i-1,1})^{y_i^{\mathtt{A}}} \right]
\end{aligned}
$$

For a PYAG, the two-stage generative process outlined above is redundant. The distributions associated with particular lexical items are trivial—they put all their mass on a single expression. In the next section we will relax this assumption, to define fragment grammars.

```
(define D (let ((sample-lexical-item
                 (PYmem a^D b^D
                        (lambda ()
                          (expand-lexical-item
                           (multinomial
                            (list (terminal "the")
                                  (terminal "a"))
                            (list θ1^D θ2^D)))))))
            (lambda () (map sample (sample-lexical-item)))))

(define N (let ((sample-lexical-item
                 (PYmem a^N b^N
                        (lambda ()
                          (expand-lexical-item
                           (multinomial
                            (list (terminal "chef")
                                  (terminal "soup")
                                  (terminal "omelet"))
                            (list θ1^N θ2^N θ3^N)))))))
            (lambda () (map sample (sample-lexical-item)))))

(define V (let ((sample-lexical-item
                 (PYmem a^V b^V
                        (lambda ()
                          (expand-lexical-item
                           (multinomial
                            (list (terminal "cooks")
                                  (terminal "works")
                                  (terminal "makes"))
                            (list θ1^V θ2^V θ3^V)))))))
            (lambda () (map sample (sample-lexical-item)))))

(define A (let ((sample-lexical-item
                 (PYmem a^A b^A
                        (lambda ()
                          (expand-lexical-item
                           (multinomial
                            (list (terminal "diligently"))
                            (list θ1^A)))))))
            (lambda () (map sample (sample-lexical-item)))))

(define AP (let ((sample-lexical-item
                  (PYmem a^AP b^AP
                         (lambda ()
                           (expand-lexical-item
                            (multinomial
                             (list (list A))
                             (list θ1^AP)))))))
             (lambda () (map sample (sample-lexical-item)))))

(define NP (let ((sample-lexical-item
                  (PYmem a^NP b^NP
                         (lambda ()
                           (expand-lexical-item
                            (multinomial
                             (list (list D N))
                             (list θ1^NP)))))))
             (lambda () (map sample (sample-lexical-item)))))

(define VP (let ((sample-lexical-item
                  (PYmem a^VP b^VP
                         (lambda ()
                           (expand-lexical-item
                            (multinomial
                             (list (list V AP)
                                   (list V NP))
                             (list θ1^VP θ2^VP)))))))
             (lambda () (map sample (sample-lexical-item)))))

(define S (let ((sample-lexical-item
                 (PYmem a^S b^S
                        (lambda ()
                          (expand-lexical-item
                           (multinomial
                            (list (list NP VP))
                            (list θ1^S)))))))
            (lambda () (map sample (sample-lexical-item)))))
```

Figure 18: Two-stage version of a PYAG.

7 Fragment Grammars

Fragment grammars (FGs) are a generalization of PYAGs which allow the distributions associated with individual lexical items to be non-trivial. There are many ways in which to do this. FGs adopt an approach which draws on the linguistic intuition associated with the idea of a heterogeneous lexicon.

If lexical items are distributions over expressions, then (partial) tree fragments with variables at their leaves can be thought of as a special kind of distribution over expressions: distributions which have been concentrated around expressions whose parse trees share the same *tree prefix*. A tree prefix is a partial tree fragment at the top of a parse tree. The way in which these kinds of tree fragments concentrate a distribution is shown in figure 19.

Fragment grammars have a two-stage generative model identical to the two-stage PYAG discussed above. First, we sample a lexical item, which in this case corresponds to the distribution associated with a tree prefix, and then from this lexical item we sample an expression. The second part of this process corresponds to finishing the derivation from that tree prefix. To define

Figure 19: Creating a table in a FG can be thought of as concentrating a distribution around a distribution over trees that have the same tree prefix (starting tree). In this case, creating a table corresponding to the tree fragment `NP` $\longrightarrow \langle$ a `N` $\rangle$ can be thought of as concentrating the `NP` distribution into distribution: `(lambda () (map sample` $\langle \langle$ a $\rangle$ `N` $\rangle$ `))`.

this generative model requires only a small change to the lexical item sampling procedures from the two-stage PYAG. This is shown in Figure 20.

```
(define (grow-child-or-not child)
    (if (flip)
        (let ((value (sample child))) (lambda () value))
        child))

(define (expand-lexical-item right-hand-side)
    (map grow-child-or-not right-hand-side))
```

Figure 20: Helper procedures for generating lexical items in a fragment grammar.

Procedure `grow-child-or-not` is a higher-order procedure which takes another procedure, `child`, as an argument. It flips a coin, and if it is comes up heads it samples from `child`, wraps the return value up as a procedure and returns it. Otherwise, it simply returns the procedure itself. In other words, it either returns a δ-distribution concentrated on some return value of `child`, or leaves `child` as is and returns it. It represents a mixture distribution over the set of δ-distributions on each possible expression returned by `child` and the distribution defined by `child` itself. The procedure `expand-lexical-item` works like before; it takes the RHS of a CFG rule, expressed as a list of procedures, and applies `grow-child-or-not` to each element of that list. Aside from these changes, the Church code for the fragment grammar and the two-stage PYAG are identical.

However, substituting `grow-child-or-not` for `sample-delta-distribution` results in a radical change of the behavior of the model. When `grow-child-or-not` decides to recurse, then the tree prefix of the corresponding lexical item grows. However, when it does not recurse, then it leaves a variable in the corresponding right-hand-side position of the lexical item.

The decision to grow the lexical item or not is made independently for each nonterminal on the RHS of a CFG rule. For simplicity, in the code above, this was accomplished by flipping a fair coin. In the actual fragment grammar implementation, we put a beta prior on the probability of this flip and integrated out the weight for each nonterminal on the right-hand-side of each rule. Doing inference over this representation allowed us to learn, for example, that a particular category in a particular position on the RHS of a rule was likely to expand when creating a new lexical item—or was likely not to.

Formally, a fragment grammar is a 4-tuple $\mathcal{F} = \langle G, \Pi, \langle a^{\mathtt{A}}, b^{\mathtt{A}} \rangle, \Psi \rangle$ where G is a context free grammar, Π are the vectors of multinomial-Dirichlet pseudo-counts for each nonterminal. $\langle a^{\mathtt{A}}, b^{\mathtt{A}} \rangle$ is the set of Pitman-Yor hyperparameters for each nonterminal, and Ψ is the set of pseduocounts for the beta-binomial distributions associated with the nonterminals on the RHSs of the rules. Let $\boldsymbol{X}$ and $\boldsymbol{Y}$ be sets of count vectors for underlying rules, and lexical items as before. Let $\boldsymbol{Z}$ be the set of count vectors counting the number of times that each nonterminal on the RHS of a CFG rule was expanded in producing a lexical item. We will use the shorthand $\boldsymbol{F} = \langle \boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z} \rangle$ for the sufficient counts representing a fragment grammar state. The joint probability of a corpus $\boldsymbol{E}$ together with parses for those expressions $\boldsymbol{P}$ is:

$$
\begin{aligned}
P(\boldsymbol{E}, \boldsymbol{P} | \mathcal{F}) &= \mathbf{fg}(\boldsymbol{F}; \mathcal{F}) \\
&= \prod_{\mathtt{A} \in V} \Bigg[\frac{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})}{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}} + x_i^{\mathtt{A}})} \frac{\Gamma(\sum_{i=1}^{K^{\mathtt{A}}} \pi_i^{\mathtt{A}})}{\prod_{i=1}^{K^{\mathtt{A}}} \Gamma(\pi_i^{\mathtt{A}})} \frac{[b^{\mathtt{A}} + a^{\mathtt{A}}]_{N^{\mathtt{A}}-1, a^{\mathtt{A}}}}{[b^{\mathtt{A}} + 1]_{K^{\mathtt{A}}-1,1}} \prod_{i=1}^{K^{\mathtt{A}}} ([1 - a^{\mathtt{A}}]_{i-1,1})^{y_i^{\mathtt{A}}} \\
&\qquad \times \prod_{r \in R^{\mathtt{A}}} \left(\prod_{\mathtt{B} \in (\mathtt{rhs}\ r)} \frac{\Gamma(\psi_{\mathtt{B}} + z_{\mathtt{B}}) \Gamma(\psi_{\mathtt{B}}^{'} + (x_r - z_{\mathtt{B}}))}{\Gamma(\psi_{\mathtt{B}} + \psi_{\mathtt{B}}^{'} + x_r)} \frac{\Gamma(\psi_{\mathtt{B}} + \psi_{\mathtt{B}}^{'})}{\Gamma(\psi_{\mathtt{B}}) \Gamma(\psi_{\mathtt{B}}^{'})} \right) \Bigg]
\end{aligned}
$$

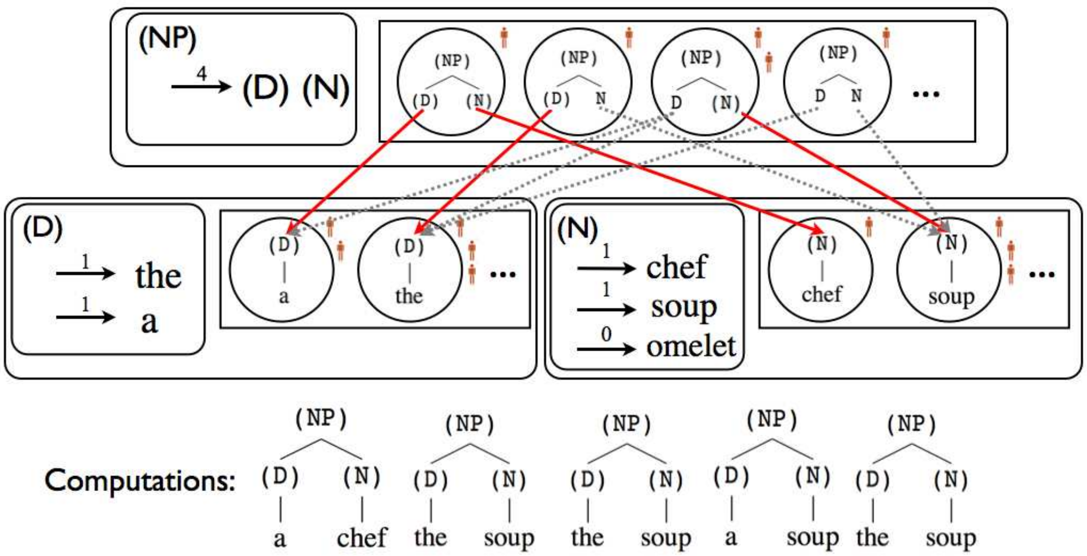

Figure 21: Representation of a possible state of a fragment grammar after having computed the five expressions shown at the bottom. Red lines represent recursions during the sampling of lexical items. They show computations that were stored inside of a lexical item. Grey dashed lines represent recursions during the sampling of expressions from lexical items. These represent computations whose result was not stored by the system.

7.1 Fragment Grammar State

We can represent a fragment grammar state in a similar way to an adaptor grammar state as shown in Figure 21. Here we show a possible state of a fragment grammar after having generated five expressions. The red lines represent recursions made by the `grow-child-or-not` procedure. These recursions resulted in larger lexical items. In particular, the `NP` restaurant contains four lexical items *1.)* ⟨ a chef ⟩ *2.)* ⟨ the `N` ⟩ *3.)* ⟨ `D` soup ⟩ *4.)* ⟨ `D N` ⟩ .

The dotted grey lines represent recursions performed to sample expressions from these lexical items. For example, the third table represents the lexical item ⟨ `D` soup ⟩. It has two customers seated at it. This lexical item was drawn once, when it was created, and then used twice to produce two different expressions: ⟨ a soup ⟩ and ⟨ the soup ⟩.

Note that, as with adaptor grammars, once a lexical item has been created, the choices which were made internal to it need never be made again. Because these choices need not be made again, they can be reused without cost. On the other hand, if an expression is drawn from a lexical item, then all the choices that are made "outside" of that lexical item—i.e., by calling a leaf procedure—must be paid every time that the lexical item is used. In other words, grey lines lead to the seating of new customers at the tables that they point to, while red lines represent structure which is free.

8 Inference

The fundamental inference question for fragment grammars is: what set of lexical items best accounts for some observed data. In other words, given (hyperparameters for) a fragment grammar $\mathcal{F}$ and a corpus of expressions $\boldsymbol{E}$, we would like to estimate the posterior distribution over lexical items used to build $\boldsymbol{E}$. Since the process of generating lexical items is the same as the process of generating expressions, the set of lexical items can be represented by the set of parses for $\boldsymbol{E}$, $\boldsymbol{P}$. We are interested in the following posterior distribution.

$$P(\boldsymbol{P}|\boldsymbol{E},\mathcal{F}) \propto P(\boldsymbol{E}|\boldsymbol{P},\mathcal{F})P(\boldsymbol{P}|\mathcal{F}) \tag{10}$$

In a following section we will describe a Metropolis-Hastings algorithm for fragment grammar inference based on those in [21, 22]. First, however, we will develop some intuitions for what kinds of fragment grammar posterior states are good and bad hypotheses about different kinds of data.

8.1 Intuitions

A fragment grammar lexical item grows when `grow-child-or-not` recurses during the sampling of a lexical item. Above we discussed the tradeoff between the number of lexical items to store and the number of choices that must be made to generate an expression. To sharpen intuitions about the optimal behavior of a fragment grammar applied to a particular data set, it is useful to consider a specific case in detail. This will allow us to see where it makes sense for the system to grow larger lexical items, and where it does not.

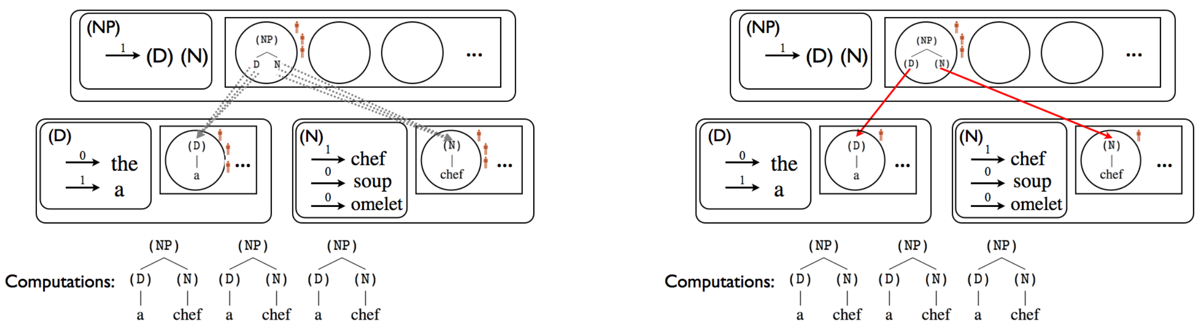

Figure 22: When it makes sense to have a more specific lexical item: repeated substructure in expressions can lead to significant savings by creating larger lexical items.

Figure 22 shows two possible fragment grammar states after having generated the expression ⟨ a chef ⟩ three times. The left hand side of the figure shows one possible extreme. Here, `grow-child-or-not` has decided for both elements of the RHS not to grow the lexical item further. The resulting lexical item is completely abstract and corresponds in form to the underlying CFG rule. To account for the data set, the grammar has seated three customers in the `NP` restaurant, as well as three customers in both the `D` and `N` restaurants.

The right hand side of the figure shows another extreme solution. In this case `grow-child-or-not` has decided to recurse the maximal amount, creating a lexical item that corresponds to the full expression ⟨ a chef ⟩. Subsequent calls to `NP` can produce this entire expression simply by seating another customer at that same table.

The Pitman-Yor process assigns higher probability to seating arrangements that minimize the number of customers and tables in a restaurant. They also assign higher probability to seating arrangements that make the restaurant "clumpier" by seating more customers at fewer tables. This example shows that when expressions repeat themselves, there can be significant savings in creating larger lexical items by recursing through `grow-child-or-not`. Creating a larger lexical item at the `NP` level in Figure 22 allowed four fewer customers to be seated at the `D` and `N` levels.

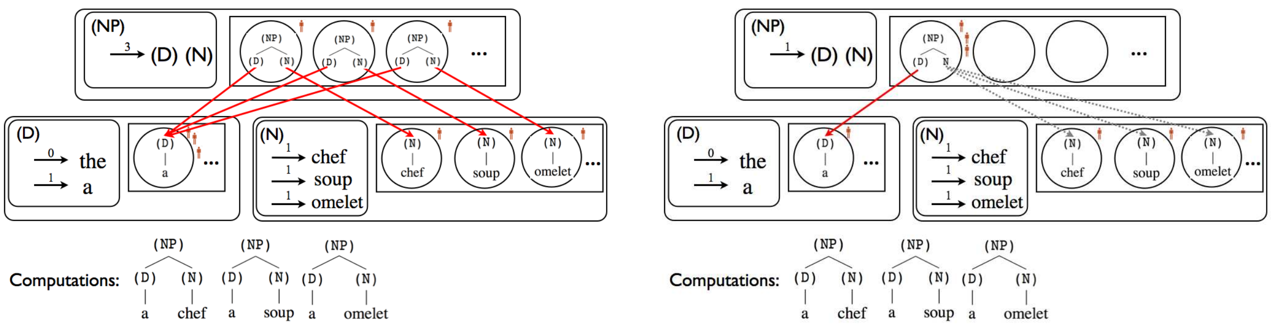

Figure 23: When it makes sense to have a lexical item with a variable.

Figure 23 shows a case where increasing the size of lexical items does not lead to improved posterior probabilities. In this example, there are three observed expressions: ⟨ a chef ⟩, ⟨ a soup ⟩, ⟨ a omelet ⟩.

On the left-hand-side of the figure we see a solution where maximally large lexical items have been created for each expression. Because each lexical item is maximally large, and the three expressions are non-identical, all three require their own table in the `NP` restaurant. Likewise, because three tables in the `NP` restaurant each use the table in `D`, that table has three customers seated at it.

On the right-hand side we see another possible posterior state. Here, `grow-child-or-not` has grown the `D` child of `NP` but not the `N` child. Because the lexical item at the table in `NP` has expression ⟨ a `N`⟩, it can be reused to generate all three expressions. Since the Pitman-Yor process prefers fewer tables for the same number of customers, the posterior score of this `NP` restaurant will be higher than that on the left. Furthermore, we only needed to seat a single customer in the `D` restaurant, the first time we built the `NP` table.

This example shows that when there is high type variability in a structure, it makes sense to have more abstraction in those positions with the variability. In cases where there is a variety of unshared, or unrepeated structures, it is better to pay the cost for the differences as few times as possible. Creating memoized lexical items allows us to pay the cost of repeated structures "higher up" in our restaurants. However, if we have distinct structures (types), it makes sense to pay the cost for these as close to the source of the distinctness as possible. Each distinct structure must be represented distinctly. If we allow this to happen in higher restaurants we will suffer from the combinatorial explosion of possibilities. It is better to "quarantine" variability as low as possible.

8.2 A Metropolis-Hastings sampler

We now describe a Metropolis-Hastings style algorithm for fragment grammar inference. The algorithm is based on those in [22, 21]. Note that this algorithm relies on several standard techniques in computational linguistics. Most important of these are the computation of the inside table for a PCFG, and sampling a parse for a sentence given this inside table. Each of these techniques has its own complications and we will not describe them in detail here. Readers seeking more details on these steps are referred to the following sources [22, 21, 16].

Given a corpus of expressions $\boldsymbol{E}$, we wish to do inference on the posterior distribution over the set of (unobserved) lexical items which gave rise to that corpus: $\boldsymbol{P}$. In general we would like to sample from $P(\boldsymbol{P}|\boldsymbol{E},\mathcal{F})$. Pitman-Yor processes, multinomial-Dirichlet distributions, and beta-Binomial distributions are all exchangeable, which means that we are free to treat any expression $e^{(i)} \in \boldsymbol{E}$ in as if it were the last expression sampled during the creation of $\boldsymbol{E}$. Our sampling algorithm leverages this fact by (re-)sampling each $p^{(i)} \in \boldsymbol{P}$ for each expression in turn.

After initializing $\boldsymbol{P}$ we loop through the set of expressions in $\boldsymbol{E}$ and for each expression $e^{(i)}$ we remove $p^{(i)}$ from our fragment grammar state $\boldsymbol{F}$.[16] We will write the new state as $\boldsymbol{F}_{-p^{(i)}}$.

Ideally, we would now like to sample a parse for $e^{(i)}$ conditional on all of the other parses so far, as if it were the last expression generated by the model. That is, we would like to implement a *Gibbs sampler*. Unfortunately, for reasons discussed below, it seems to be impossible to exactly sample from this Gibbs distribution efficiently. Instead, we define an approximating PCFG $G'(\boldsymbol{F}_{-p^{(i)}},\mathcal{F}))$, and sample a new analysis from this.

[16]In particular we remove `(counts` $p^{(i)}$`)`

$$
\begin{aligned}
p^{(i)\prime}|e^{(i)}, \boldsymbol{F}_{-p^{(i)}} &\sim P(\cdot|e^{(i)}, \boldsymbol{F}_{-p^{(i)}}, \mathcal{F}) \\
&\approx P(\cdot|e^{(i)}, G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F})) \\
&= \mathbf{pcfg}(\cdot; G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F}))
\end{aligned} \tag{11}
$$

This serves as a proposal distribution for our MH algorithm. We then accept this proposal with probability $\mathcal{P}(p^{(i)}, p^{(i)\prime})$, the MH criterion.

$$
\mathcal{P}(p^{(i)}, p^{(i)\prime}) = min\left\{1, \left[\frac{\mathbf{fg}(\boldsymbol{F}_{-p^{(i)},+p^{(i)\prime}}; \mathcal{F})}{\mathbf{fg}(\boldsymbol{F}; \mathcal{F})} \times \frac{\mathbf{pcfg}(p^{(i)}; G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F}))}{\mathbf{pcfg}(p^{(i)\prime}; G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F}))}\right]\right\} \tag{12}
$$

8.2.1 The approximating PCFG: $G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F})$

As discussed earlier, PCFGs make strong conditional independence assumptions. All choices made within a computation and between computations are independent. One important consequence of these conditional independence assumptions is that there are efficient dynamic programming algorithms available for solving the PCFG parsing problem. These algorithms rely on the fact that the distribution over parses for an expression e is a simple function of distributions over parses for subexpressions of e.

This is not the case for MD-PCFGs, PYAGs and FGs. To see this, observe that any choice made in one of these formalisms immediately changes the probabilities of all the other possible choices for the same nonterminal procedure. If the same procedure is invoked several times in a parse, then the probabilities for choices made by each invocation are not independent from one another.[17] Chasing down these dependencies for exponentially many parses destroys the time-bounds of parsing algorithms.

To address this problem we define an approximating PCFG $G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F})$, which we will use to construct a proposal distribution for our MH sampler. The idea behind the approximation is that after observing a number of expressions, the amount by which one more use of a particular lexical item or base rule will change the overall distribution is minimal. For example, if the same lexical item is used twice in a parse, then the probability of all the parses in the chart should be adjusted to account for the fact that these uses are not independent. However, if this lexical item is one of thousands in a restaurant, then this non-independence will only have a small effect of the overall distribution. Thus

[17] They are exchangeable, however.

we can approximate the correct distribution over parses by pretending that all lexical item uses are independent. In effect we "freeze" all the restaurants associated with nonterminals in our grammar, holding their counts constant while we parse the next expression. This frozen grammar represents our fragment grammar at an instantaneous snapshot in time. This snapshot is a PCFG. We then parse with this PCFG and use the trees it provides as proposals for our MH algorithm.

In fact, when we sample from the approximating PCFG, the first decision we make is correct from the point of view of the FG we were approximating. Because we do not update the corresponding counts as we sample, however, the approximation becomes progressively worse over the course of the parse.

Intuitively, a lexical item $\mathtt{v}^{\mathtt{A}}$ can be thought of as a context-free rule $\mathtt{A} \longrightarrow \mathtt{v}$. For example, a lexical item in the `NP` restaurant corresponding to the expression $\langle$ a `N` $\rangle$ can be thought of as the CFG rule $\mathtt{A} \longrightarrow \langle \text{ a } \rangle\ \mathtt{N}$. We can merge lexical items with the set of rules from the underlying grammar to create the set of rules for our approximating PCFG. For each possible sequence of terminals and nonterminals, γ, found at the leaves of a lexical item in our grammar, or on the RHS of an underlying rule we add a rule to our approximating grammar, $\rho^{\mathtt{A}}_{\gamma} = \mathtt{A} \longrightarrow \gamma$. The probability of this rule is given as follows.

$$
\begin{aligned}
\theta_{\rho^{\mathtt{A}}_{\gamma}} = & \sum_{\mathtt{v} \in \mathtt{A} | (\mathtt{rhs}\ l) = \gamma} \frac{y^{\mathtt{A}}_{\mathtt{v}} - a^{\mathtt{A}}}{N^{\mathtt{A}} + b^{\mathtt{A}}} \\
& + \sum_{r \in R^{\mathtt{A}} | (\mathtt{rhs}\ r) = \gamma} \left[\frac{K^{\mathtt{A}} a^{\mathtt{A}} + b^{\mathtt{A}}}{N^{\mathtt{A}} + b^{\mathtt{A}}} \times \frac{x^{\mathtt{A}}_{r} + \pi^{\mathtt{A}}_{r}}{K^{\mathtt{A}} + \sum \vec{\pi}^{\mathtt{A}}} \right]
\end{aligned}
$$

The first term adds in the probability of all lexical items in the `A` restaurant which have γ as their sequence of leaves. The second term adds in the probability of all underlying CFG rules with `A` on their LHS which share γ as their RHS. The second term has two components; the first represents the probability of sitting at a new table in the `A` restaurant. The second represents the probability of choosing the base rule as the label for that new table. Thus we marginalize over all lexical items and underlying CFG rules which have the form $\mathtt{A} \longrightarrow \gamma$.

Given this approximating grammar we can efficiently compute the distribution over parses given some expression. To turn this into a proposal distribution $P(p^{(i)'} | e^{(i)}, G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F}))$, we do the following.

1. Compute the inside table over parses of $e^{(i)}$ given $G'(\boldsymbol{F}_{-p^{(i)}}, \mathcal{F}))$. This table includes the inside probability of every possible constituent in ev-

ery possible parse of $e^{(i)}$. We compute this with a version of the CYK algorithm.

2. Sample a parse using the inside table. Starting at the goal item, we sample a parse for $e^{(i)}$ by recursively sampling from the distribution over backpointers resulting from normalizing inside scores. This algorithm is described in more detail in [22].

3. Sample an FG parse $p^{(i)'}$ using the approximating parse. The rules used in the approximating grammar collapse across fragment grammar lexical items and underlying CFG rules. We recover a FG parse by sampling conditionally on the collapsed parse, undoing the marginalization we performed when we calculated the probability of rule ρ^{A}. We do this bottom-up along the structure of the approximating grammar parse. We create a new table each time that an approximating parse node corresponds to a underlying CFG rule in our fragment grammar.

Once we have sampled a proposal parse, we calculate our MH score and accept or reject accordingly.

8.3 Implementation

The MH sampler just described has been implemented in the OCAML programming language and can be made available upon request to the first author.

9 Preliminary Evaluation on the Switchboard Corpus

As a preliminary step in evaluating the fragment grammar model, we explored reuse in a corpus of natural language utterances: the Switchboard corpus of spoken English [13]. Our Bayesian model's basic tradeoff is between the number of fragments which must be stored (and relatedly the amount that fragments can be reused) and the number of choices that have to be made to generate any single sentence. If we store only small, abstract fragments, we will be able to reuse them in many sentences and therefore we will be able to get by with fewer lexical items. However, generating individual sentences will require many independent choices. On the other hand, if we store large, concrete fragments of structure, the number of choices that we need to make to generate a single sentence will be smaller, but we will need to store many more fragments in memory and each structure will be reused less often in the corpus.

We examine our corpus using two measures which directly reflect the nature of this tradeoff. First, we look at the *reuse* of stored items. The reuse of a lexical item is the number of times that the item was used in the corpus. We expect that as items become smaller and more abstract, their reusability will increase. Below, using the Switchboard corpus, we will compare reuse rates under the fragment grammar posterior with the rate of reuse under a minimal storage regime where only minimal tree fragments are stored.

Second, we look at how many independent choices were needed on average to generate each sentence. This measure is the *percent choices* per sentence. Out of all the parse tree nodes in a particular parse tree, how many of them were made as independent choices as opposed to being internal to some lexical item? Below we will compare the average number of choices needed per sentence under the fragment grammar posterior with the same measure under a maximal storage regime—where only maximal, complete utterance-sized fragments have been stored.

9.1 Method

Input Data Switchboard is a corpus of spontaneous telephone conversations [13], part of which has been annotated for part-of-speech and hierarchical syntactic structure, amongst other information [28]. Prior to using the corpus, we transformed the annotation, removing markup relating to dysfluencies and other speech errors. Furthermore, we imposed binary branching on all syntactic structure and projected a simple X-bar style grammar from part-of-speech tags and phrasal heads found using the Stanford parser's headfinder [25].

After transformation, the corpus contained 78,838 sentences of average length 9.16 words.

Simulation We ran our Metropolis-Hastings algorithm for a total of 75 sweeps[18] through the corpus with the following parameter settings: G: A CFG was read from the trees in the Swtichboard corpus; $a = 0$ and $b = 1$; $\pi = 1$; $\theta = .5$ and $\psi = 100$. Parses were initialized randomly, independent of one another. Individual sentence samples were conditioned on the parse trees from the corpus. The results reported in the next section refer to the final, 75th, sample taken.

[18]Data reported in the left half of Figure 24 is the result of fewer (6) sweeps. This is due to a bug that forced us to restart the sampler run.

9.2 Results and Discussion

The goal of our simulation study was to investigate the tradeoff between computation and reuse in a natural language corpus, as reflected in the fragment grammar analyses found for the corpus.

In Figure 24, left, we have given a histogram showing reuse in the analyses found by our fragment grammar simulation. For comparison, on the right we include a similar histogram for a model equivalent to the underlying PCFG—that is, a model with the minimal storage regime in Figure 1.

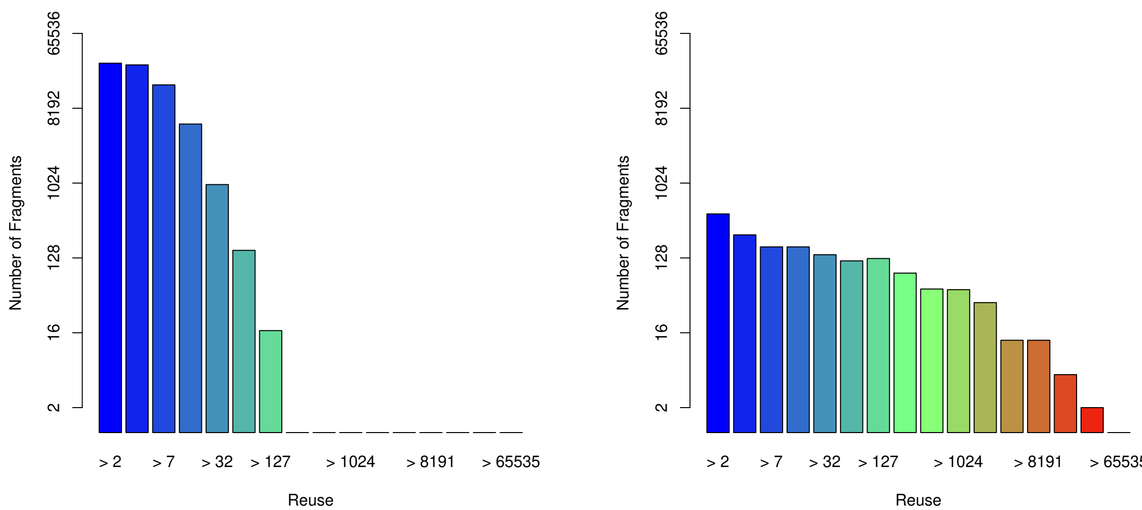

Figure 24: This figure shows the distribution of reuse rates amongst the stored fragments in the grammar. On the left is the result of fragment grammar simulation. On the right is the result for a grammar with a minimal storage regime.

The fragment grammar has significantly lower rates of reuse than the underlying PCFG. This implies that the fragment grammar has learned a variety of medium-sized fragments from the corpus and is using significantly more memory resources in storing these fragments than does the PCFG. What is gained in return for this additional memory use is that fewer independent choices are required—each choice "frozen" inside a fragment is a choice that the PCFG would have to make independently.

In Figure 25 we show the independent choice results for our simulation. On the left is a histogram describing the number of independent choices made, across sentences, for our fragment grammar simulation. For comparison, on the right we provide the independent choice results of a model corresponding to the maximal storage regime from Figure 1. The maximal storage regime is strongly peaked at low percentages because there can be only one choice made per sentence under that model; but it pays the cost of needing to store every sentence as a fragment in its entirety. Fragment grammar, by contrast, makes many more choices on average to generate each sentence—another consequence of storing mid-sized chunks.

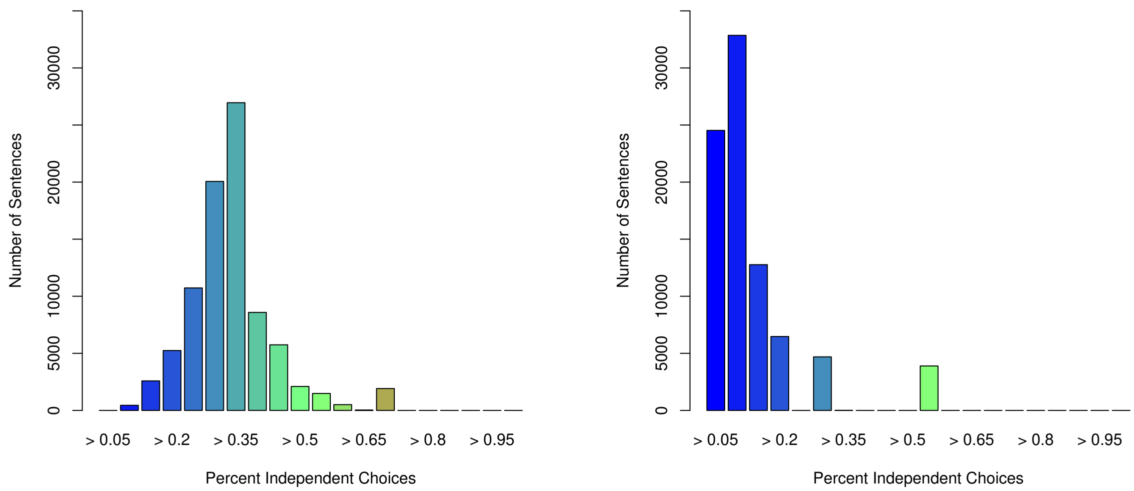

Figure 25: Results for independent choice percent for fragment grammar run. On the left is the fragment grammar data. On the right are the results for a grammar with a maximal storage regime.

10 Experimental Data: Artificial Language Learning

While corpus analyses are a useful tool for exploring the statistical characteristics of samples of natural language, they cannot tell us about the psychological plausibility of our model for real human learning. To test the model's psychological plausibility, we ran an *artificial language learning* (ALL) experiment (see e.g. [29, 9]).

In the ALL paradigm, subjects are first exposed to sequences of nonsense stimuli constructed according to some pattern. Then they are tested to see if they were able to learn the pattern. The ALL paradigm allows us to directly manipulate the stimulus statistics which we hypothesize lead to storage and reuse of bigger and smaller structures.

We wish to manipulate the fragments of structure stored by participants. In our present context, this means whether or not participants stored associations between artificial language syllables and the positions in which they appeared. Figure 26 illustrates the basic idea of the design. Stimuli were short, two-syllable sequences, AB. We manipulated the frequency with which items appeared in position A or position B while keeping the total number of exposure stimuli constant. Figure 26 shows three idealized exposures. In the top row, there is only one item which appears in the A position, while there are four different items which appear in the B position. Under our model, we hypothesize that this should lead to storage of a fragment with a_1 in the first position. The symmetric case in the B position is shown in the middle row.

Our test items consisted of sequences in which either the item appearing in the A position **or** the item appearing in the B position was completely novel. This is shown on the right-hand side of Figure 26. When a novel item appears in a position, the participant has no choice but to generate a structure for it from scratch. The test item which contains the novel B in the first row can be generated using the fragment which links a_1 to the first position. This is a highly reused fragment, and thus has high probability in the memoizer. Using this fragment also means fewer independent choices are required to generate the test stimulus. On the other hand, the test item which contains the novel A item can only reuse the smaller, less frequent fragment containing b_1. Generating this sequence also requires more independent choices. We hypothesize that when asked to choose which test item is more likely to be from the "same" language as the exposure stimuli, subjects will favor the one that represents more reuse, and fewer independent choices.

The bottom row of Figure 26 shows the predictions when there is no asymmetry between A and B position. Under this condition the smaller fragments containing a_1 and b_1 are both more frequent and have a higher probability in the memoizer than they did in the other conditions. Moreover, both test items require an equal number of independent choices. In this case we hypothesize subjects will be at chance in choosing between the test stimuli.

10.0.1 Method

Design There were 9 between-subject A:B ratios: (1:36), ..., (4:9), (6:6), ..., (18:2), (36:1). In addition to the critical tests discussed above, we included

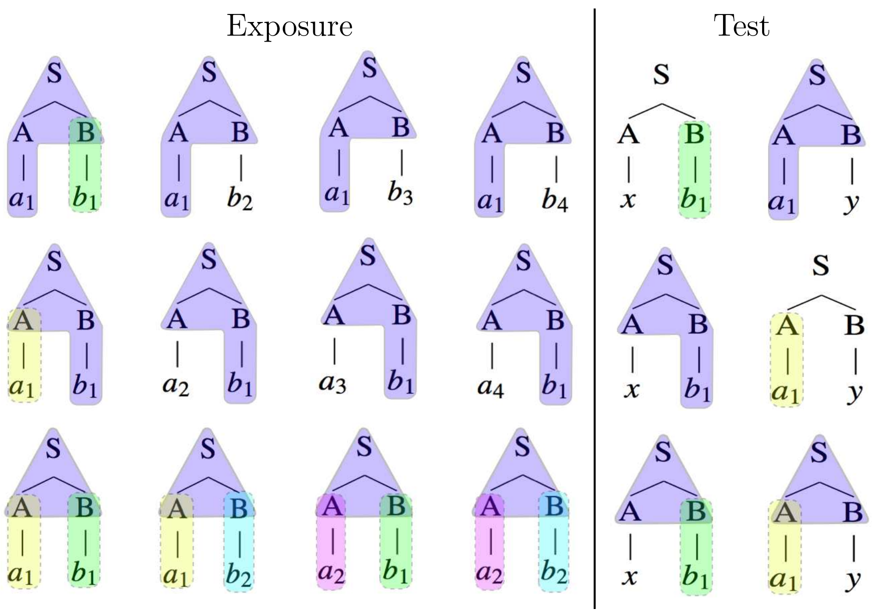

Figure 26: Experimental setup. The left-hand side shows possible reuse in three idealized between-subject exposures. The right hand side shows how the reusable fragments apply to critical test items.

additional test items of the form $[a_1\ b_1]$ v. $[x\ b_1]$ in order to rule out the hypothesis that participants were simply forgetting items that appeared during exposure.

Procedure The experiment was conducted through a web browser using software described at `www.astoundment.com/netlab/`. The experiment consisted of 2 blocks where participants first listened to the exposure stimuli for 3 repetitions and then went on to perform 15 forced choice test trials; 5 critical tests and 10 controls.

Modeling We created simulation data by running the fragment grammar sampler on the simple positional PCFG described above. For each condition, we ran 24 simulation runs under a variety of a, b, θ, and ψ parameter settings. Each simulation ran for a total of 10,000 sweeps through the 300 training sequences. After each sweep, we scored the training sequences and test sequences described above. These scores were averaged across sweeps and across simulations for each condition. Note that by averaging across a range of parameter settings our results are independent of the parameters of the model.[19]

To produce an experimental prediction from our simulations, it was necessary to turn our sequence scores into choice predictions. We did this using the Luce choice rule $P(c_i) \propto p(s_i)^{\beta}$. Where $P(c_i)$ is the probability of choosing sequence s_i, $p(s_i)$ is the probability of sequence s_i and β is a parameter which controls how much the better option is preferred. We took $\beta = 0.5$.

Participants Data was collected both online and in the lab using the web-based software described above. Participants were excluded if they did not complete at least 15 overall test trials and 5 critical (half) test trials. 308 participants met the criteria.

10.0.2 Results and Discussion

Figure 27 shows the results of both the participant (black) and simulation (red) data. Along the x-axis is the ratio of A to B items. Along the y-axis is the percent of the time that participants chose the stimulus with a novel A item in the critical tests. Correlation between simulation and test data was high ($r = 0.97$).

Participants showed a bias in all conditions to favor new items in the A position. This may be an artifact of the fact that most of our subjects were

[19]This also explains the high variance in the simulation means.

English speakers, and English inflectional morphology tends to be suffixal. No such prior bias was built into the simulation.

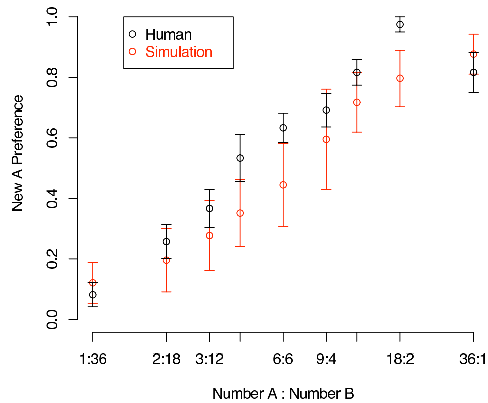

Figure 27: Results of experiment. X-axis shows the ratio of A position items to B position items. The y-axis shows the proportion of the time that participants chose the stimulus with a new A position item in the crucial tests.

11 Relation to Other Models in the Literature.

The modeling work which is most similar to the work reported here is that of Data Oriented Parsing (DOP) [3, 4, 39]. DOP, in its original form, is built on the formalism of tree substitution grammar, a generalization of PCFGs which allows the basic building blocks of the grammar to be arbitrary tree fragments. DOP provides a framework for estimating the probabilities of these tree fragments from corpora. DOP has also been extended to tree-adjoining and lexical-functional grammars (see articles in [3]).

In a certain sense, DOP is very similar to the model presented here. Both embrace the idea that the set of lexical items can consist of a large number of heterogeneous tree structures. The main difference between the work presented here and DOP is that the present system provides a full generative model of how the lexical items are built. In contrast, DOP takes a two part approach. First, an algorithm is applied to a dataset to learn a tree substitution grammar. Then, the resulting grammar is used to generate data.

In fragment grammar, and more generally in any generative system using stochastic memoization, the generative model itself tells us how reuse and storage are to happen in combination. In principle, every choice made during generation has an *immediate* effect on the underlying grammar, as new forms are created and old forms are reused—even during the generation of a single sentence.[20]

In a sense, the tree substitution grammar learned by a DOP model can be seen as an instantaneous snapshot of a fragment grammar state. Fragment grammar provides a fully Bayesian solution to the problem of learning the set of fragments used in DOP.

12 Conclusion

We have presented fragment grammars—a model which explores the computation/reuse tradeoff in natural language. We expressed fragment grammars using the vehicle of stochastic functional programming, and specifically the Church language [17]. The most important feature of Church from the fragment grammar perspective is its language-level support for stochastic memoization. We showed how stochastic memoization can be used to formalize the notion of reuse of computation.

The most important technical contribution of the fragment grammar model is its conceptualization of lexical items as distributions in a two-stage model. During the generation of a constituent in a fragment grammar, first a (perhaps novel) lexical item is chosen and then an expression is generated from this item. This two-stage model means that the process of learning a lexicon is integrated fundamentally into the process of using language.

We also reported the results of some preliminary empirical evaluations of the model. We found that, as predicted, the model finds intermediate sized fragments of structure in a corpus of natural language dialog. It also makes accurate predictions in an experimental, artificial language learning setting.

Acknowledgements

We would like to thank Frank Jaekel, Jelle Zuidema, Dan Roy, Jesse Snedeker, Marjorie Freedman and Sam Scarano for detailed comments on drafts of this report. We would also like to thank Dan Roy and Mark Johnson for many clarifying discussions about the model.

[20]As discussed above, our inference algorithm makes use of an approximation which does not update lexical item counts during the parsing of a single sentence.

References

[1] Jean-Michel Autebert, Jean Berstel, and Luc Boasson. Context-free languages and pushdown automata. In *Handbook of Formal Languages*, volume 1, pages 111–172. Springer-Verlag, 1997.

[2] Laurie Bauer. *Morphological Productivity*. Cambridge University Press, 2001.

[3] Rens Bod, Remko Scha, and Khalil Sima'an, editors. *Data-Oriented Parsing*. CSLI, 2003.

[4] Gideon Borensztajn, Willem Zuidema, and Rens Bod. Children's grammars grow more abstract with age – evidence from an automatic procedure for identifying the productive units of language. Cogsci 2008, 2008.

[5] Noam Chomsky. *Aspects of the Theory of Syntax*. The MIT Press, Cambridge, MA, 1965.

[6] William Croft. *Radical Construction Grammar: Syntactic Theory in Typological Perspective*. Oxford University Press, 2001.

[7] Peter Culicover and Ray Jackendoff. *Simpler Syntax*. Oxford University Press, Oxford, 2005.

[8] Anna Maria Di Sciullo and Edwin Williams. *On the Definition of Word*. MIT Press, 1987.

[9] Ansgar D. Endress. Primitive computations in speech processing. (Under review), 2007.

[10] T.S. Ferguson. A bayesian analysis of some nonparametric problems. *Ann. Statist*, 1(2):209–230, 1973.

[11] Cameron E. Freer and Daniel M. Roy. Computable exchangeable sequences have computable de Finetti measures. In Klaus Ambos-Spies, Benedikt Löwe, and Wolfgang Merkle, editors, *Mathematical Theory and Computational Practice: Fifth Conference on Computability in Europe, CiE 2009*, 2009.

[12] Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin. *Bayesian Data Analysis, Second Edition*. Chapman & Hall/CRC, 2003.

[13] J.J. Godfrey, E.C. Holliman, and J. McDaniel. Switchboard: Telephone speech corpus for research and development. *IEEE ICASSP*, pages 517–520, 1992.

[14] Adele E. Goldberg. *Constructions at Work*. Oxford University Press, Oxford, 2005.

[15] Sharon Goldwater, Thomas L. Griffiths, and Mark Johnson. Interpolating between types and tokens by estimating power-law generators. In *Advances in Neural Information Processing Systems 18*, 2006.

[16] Joshua Goodman. *Parsing Inside-Out*. PhD thesis, Harvard University, 1998.

[17] Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Uncertainty in Artificial Intelligence*, 2008.

[18] Hemant Ishwaran and Lancelot F. James. Gibbs sampling methods for stick-breaking priors. *Journal of the American Statistical Association*, 96(453):161–173, 2001.

[19] R. Jackendoff. *Foundations of language*. Oxford University Press New York, 2002.

[20] Mark Johnson. Memoization in top-down parsing. *Computational Linguistics*, 21(3):405–417, 1995.

[21] Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Adaptor grammars: A framework for specifying compositional nonparametric bayesian models. In *Advances in Neural Information Processing Systems 19*, 2007.

[22] Mark Johnson, Thomas L. Griffiths, and Sharon Goldwater. Bayesian inference for pcfgs via markov chain monte carlo. In *Proceedings of the North American Conference on Computational Linguistics*, 2007.

[23] Daniel Jurafsky and James H. Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. MIT Press, 2000.

[24] Olav Kallenberg. *Probabilistic symmetries and invariance principles*. Springer, 2005.

[25] Dan Klein and Christopher D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics*, pages 423–430. Association for Computational Linguistics, 2003.

[26] Kenichi Kurihara and Taisuke Sato. Variational bayesian grammar induction for natural language. In *The 8th International Colloquium on Grammatical Inference (ICGI-2006)*, 2006.

[27] Chris Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Ma, May 1999.

[28] Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz, and Ann Taylor. Treebank-3. Technical report, Linguistic Data Consortium, Philadelphia, 1999.

[29] Toben H. Mintz. Category induction from distributional cues in an artificial laguage. *Memory and Cognition*, 30(5):678–686, 2002.

[30] S. Nooteboom, F. Weerman, and F. Wijnen, editors. *Storage and computation in the language faculty*. Kluwer Academic Press, 2002.

[31] Peter Novig. Techniques for automatic memoization with applications to context-free parsing. *Computational Linguistics*, 17(1):91–98, 1991.

[32] Steven Pinker. *Words and Rules*. Basic Books, 1999.

[33] Jim Pitman. Combinatorial stochastic processes. Technical report, Department of Statistics University of California Berkeley, 2002.

[34] Ivan A. Sag, Thomas Wasow, and Emily M. Bender. *Syntactic Theory: A Formal Introduction*. CSLI, 2 edition, 2003.

[35] Jayaram Sethuraman. A constructive definition of dirichlet priors. *Statistica Sinica*, 4(2):639–650, 1994.

[36] Yee Whye Teh. A hierarchical bayesian language model based on pitman-yor processes. In *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, pages 985–992, Sydney, Australia, July 2006. Association for Computational Linguistics.

[37] David Weir. *Characterizing Mildly Context-sensitive Grammar Formalisms*. PhD thesis, University of Pennsylvania, 1988.

[38] George Kingsley Zipf. *The psycho-biology of language; an introduction to dynamic philology.* Houghton Mifflin Company, Boston, 1935.

[39] Willem Zuidema. Parsimonious data-oriented parsing. In *Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL 2007)*, 2007.

A Integrating Out Parameters and de Finetti Representations

In this appendix, we discuss another way of deriving the Chinese restaurant, multinomial-Dirichlet and Pitman-Yor distributions. In our presentation above, each of these distributions was described in terms of assigning probabilities to partitions of objects. With CRPs and PYPs we were able to assign objects to partitions with potentially unbounded numbers of components. With MDD we assigned objects to partitions with a fixed number of components. In each case, we defined the partition distributions in terms of a sequential sampling scheme. In each of the cases, however, there is also an alternate construction of the distribution in terms of a hierarchical mixture model. This is known as the *de Finetti* representation of the distribution.

We illustrate this fact with the multinomial-Dirichlet distribution. Imagine a multinomial distribution over K elements with parameters specified by parameter vector $\vec{\theta}$. It is possible to draw $\vec{\theta}$ from a prior distribution rather than specifying it directly. As discussed above, the most common prior on multinomial parameters is the Dirichlet distribution [12]. Let $\vec{\pi}$ be a vector of positive real hyperparameters of length K, the Dirichlet distribution is defined as:

$$P(\vec{\theta}|\vec{\pi}) = \frac{\Gamma(\sum_{i=1}^{K} \pi_i)}{\prod_{i=1}^{K} \Gamma(\pi_i)} \prod_{i=1}^{K} [\theta_i]^{\pi_i - 1}$$

The Dirichlet distribution can be thought of as a defining a distribution over (biased) K-sided dice. To generate j multinomial observations, $\mathtt{v}^{(1)}, ..., \mathtt{v}^{(j)}$, using the Dirichlet, we first sample our K-sided die, $\vec{\theta}|\vec{\pi}$ from a Dirichlet distribution, and then use this die to sample our j observations. The joint distribution on the die, $\vec{\theta}$, and observations together is given by the following equation.

$$P(\mathtt{v}^{(1)}, ..., \mathtt{v}^{(j)}, \vec{\theta}|\vec{\pi}) = \frac{\Gamma(\sum_{i=1}^{K} \pi_i)}{\prod_{i=1}^{K} \Gamma(\pi_i)} \prod_{i=1}^{K} [\theta_i]^{\pi_i + x_i - 1} \tag{13}$$

Here the x_i refer to the counts of observed values which were equal to $\mathtt{v}_i$ (i.e. in partition component i). The generative process is shown on the left side of Figure 28.

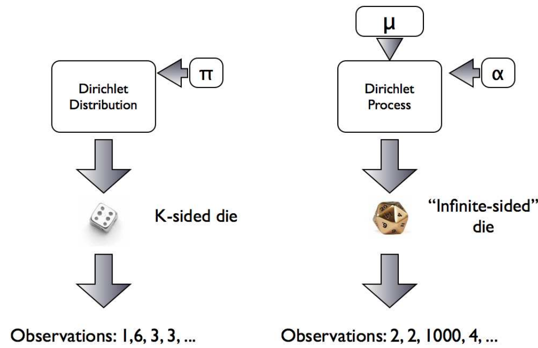

Figure 28: Graphical model representation of the generative processes for the multinomial-Dirichlet distribution and the Dirichlet process. In both cases, we first draw a die from the prior distribution, and then sample i.i.d. observations from this die. For the multinomial-Dirichlet distribution, the die is finite; for the Dirichlet process, the die is infinite.

The combination of a Dirichlet prior and a multinomial likelihood can be analytically integrated over the space of possible dice; this space is a K-dimensional *simplex*, the K-dimensional analogue of a triangle, usually written as $\vec{\theta} \in \Delta_\theta$. This leads to the closed form of the MDD which we gave above:[21]

$$
\begin{aligned}
P(\mathtt{v}^{(1)}, ..., \mathtt{v}^{(j)}|\vec{\pi}) &= \int_{\Delta_\theta} \left[\frac{\Gamma(\sum_{i=1}^K \pi_i)}{\prod_{i=1}^K \Gamma(\pi_i)} \prod_{i=1}^K [\theta_i]^{\pi_i + x_i - 1} \right] d\theta & (14) \\
&= \left[\frac{\prod_{i=1}^K \Gamma(\pi_i + x_i)}{\Gamma(\sum_{i=1}^K \pi_i + x_i)} \frac{\Gamma(\sum_{i=1}^K \pi_i)}{\prod_{i=1}^K \Gamma(\pi_i)} \right] & (15)
\end{aligned}
$$

Remember that this was the closed form expression for the probability of the Polya urn scheme representation of the MDD. In fact, the two representations—sequential sampling and hierarchical—are equivalent. They generate the same probability distributions.

[21] This integrated version of the multinomial-Dirichlet distribution is sometimes referred to as a multivariate-Polya distribution.

When we view the MDD as a hierarchical model, the individual observation draws are conditionally independent given the die $\vec{\theta}$. When the value of $\vec{\theta}$ is unobserved and must be inferred from data, we refer to this hierarchical model as the de Finetti representation of the distribution. From a Bayesian perspective, the uncertainty over $\vec{\theta}$ can be removed by integrating over possible $\vec{\theta}$s, as we did above.

Integrating in this way makes explicit the fact that $\vec{\theta}$ is an hidden variable of the model. Intuitively, when we do not know the value of $\vec{\theta}$, observations drawn from it are no longer independent. Each observation changes our posterior beliefs about the die weights. For example, if our die was two-sided—a coin—and it came up heads 25 times in a row, we would probably be inclined to believe that $\vec{\theta}$ was biased heavily in favor of heads.

The sequential construction of the MDD can be viewed as an explicit version of this. As we see more evidence that one particular value of $\vec{\theta}$ is more or less likely than the others, we predict that the next draw will favor it more or less. The distribution over observations given by this scheme is exactly equivalent to the hierarchical version where $\vec{\theta}$ is unobserved.

A similar derivation can be given for the CRP. In the case of the CRP, we do not draw finite dice, but rather we use a construction which allows us to lazily draw infinite dice. *Lazy* in this sense means that instead of eagerly computing all die weights at once, we instead draw individual weighs as we need them. This construction is called the *Dirichlet process* (DP) [10]. Integrating over possible infinite die draws from the DP leads to the CRP in the same way that integrating over possible finite die draws for the MDD leads to the Polya urn scheme representation of the MDD. [35].[22] This is shown in Figure 29.

A similar derivation can be given for the Pitman-Yor process using the *generalized Dirichlet distribution* [18]. The fact that CRPs, PYPs, and MDDs all have these dual representations follows from de Finetti's theorem. De Finetti's theorem states that a sequence of random variables is exchangeable if and only if it has a representation as a mixture with unobserved mixture weights [24]. Recently a computable version of the theorem has been proven which shows that this equivalence is not only guaranteed in principle but can be computed in practice [11].

Because all of the distributions used in this report are exchangeable, each of them will have both a de Finetti representation and a representation in terms of sequential sampling.

[22] In the version of Church presented in [17], the memoization distribution associated with the procedure `PYmem` is in fact built on the Dirichlet Process, and corresponds to lazily drawing an infinite-sided die.

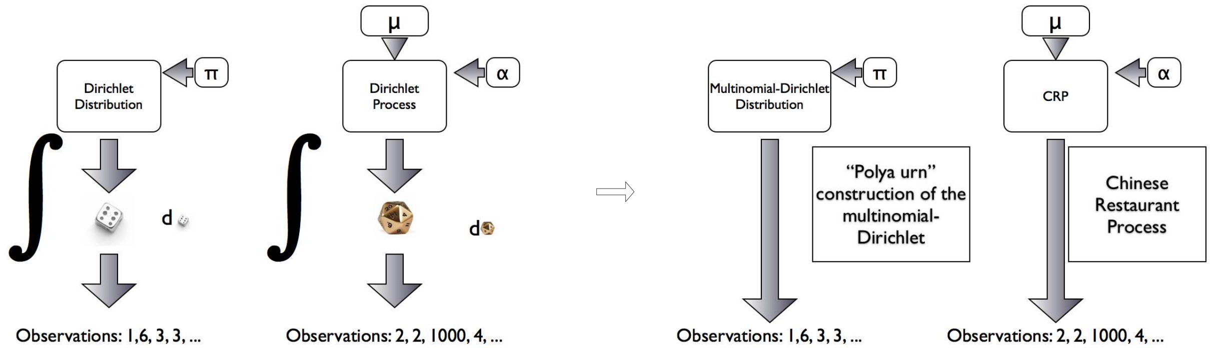

Figure 29: Relationship between the sequential sampling and de Finetti representations of the multinomial-Dirichlet distribution and Dirichlet/Chinese restaurant processes. On the left are the de Finetti representations of the two distributions. By integrating over the hidden parameters—in this case the dice weights—we derive the sequential sampling constructions.

A.1 de Finetti Representation for MD-PCFGS

In Section 3.1.3 above, we described multinomial-Dirichlet PCFGs. Now, armed with our understanding of the equivalence between the sequential sampling and de Finetti representations of the MDD, we give the code for an MD-PCFG in its de Finetti representation in Figure 30.

An important feature of this Church code is that each multinomial parameter vector $\vec{\theta}^{\mathtt{A}}$ is drawn when each procedure is first `define`d. Then the value is *closed* over inside of the `lambda` corresponding to the procedure. In other words, some value of $\vec{\theta}^{\mathtt{A}}$ is drawn once, when the procedure first comes into existence, and then a *closure* containing that value is returned as the procedure corresponding to `A`. A closure refers to a procedure that is evaluated in the context of some bound variables. In this case, the variable $\vec{\theta}^{\mathtt{A}}$.

In Church, the de Finetti representation of a distribution will often have this flavor: draw an unobserved distribution, and then return a closure containing (and conditioned on) this distribution.

We discussed in the preceding section the equivalence between de Finetti and sequential sampling representations. Although they are equivalent in terms of the distributions they define, in terms of practical inference, it is often better to use the sequential sampling representation.

Search over posterior states in the de Finetti representation involves searching over the draws of $\vec{\theta}$ inside the closure for each nonterminal. These values

```
(define D (let ((θ⃗ D (dirichlet π⃗ D )))
            (lambda ()
              (map sample
                   (multinomial
                    (list (terminal "the")
                          (terminal "a"))
                    θ⃗ D)))))

(define N (let ((θ⃗ N  (dirichlet π⃗ N)))
            (lambda ()
              (map sample
                   (multinomial
                    (list (terminal "chef")
                          (terminal "soup")
                          (terminal "omelet"))
                    θ⃗ N)))))

(define V (let ((θ⃗ V  (dirichlet π⃗ V)))
            (lambda ()
              (map sample
                   (multinomial
                    (list (terminal "cooks")
                          (terminal "works")
                          (terminal "makes"))
                    θ⃗ V )))))

(define A (let ((θ⃗ A (dirichlet π⃗ A)))
            (lambda ()
              (map sample
                   (multinomial
                    (list (terminal "diligently"))
                    θ⃗ A)))))

(define AP (let ((θ⃗ AP (dirichlet π⃗ AP)))
             (lambda ()
               (map sample
                    (multinomial
                     (list (list A))
                     θ⃗ AP)))))

(define NP (let ((θ⃗ NP (dirichlet π⃗ NP)))
             (lambda ()
               (map sample
                    (multinomial
                     (list (list D N))
                     θ⃗ NP)))))

(define VP (let ((θ⃗ VP (dirichlet π⃗ VP)))
             (lambda ()
               (map sample
                    (multinomial
                     (list (list V AP)
                           (list V NP))
                     θ⃗ VP)))))

(define S (let ((θ⃗ S (dirichlet π⃗ VP)))
            (lambda ()
              (map sample
                   (multinomial
                    (list (list NP VP))
                    θ⃗ S)))))
```

Figure 30: The de Finetti representation of a multinomial-Dirichlet probabilistic context-free grammar.

are continuous, and search over them can be a costly process. In the sequential sampling representation, we can do inference over the number of times a particular rule was used in a corpus and then update the counts for each rule, and use the updated counts to re-score. In the de Finetti representation, inference over the draws of $\vec{\theta}$ will generally have to be implemented in separate step from search over rule uses.