mcfgcky

Kyle Grove
Linguistics Department
Cornell University
kwg33@cornell.edu

November 19, 2010

Contents

1 Introduction and Theoretical Background

This document describes the implementation of a Multiple Context Free Grammar (MCFG) [H. Seki and Kasami, 1991, Nakanishi et al., 1997] parser, implemented in OCaML for use as a back-end of a Minimalist Grammar [Stabler, 1997] parsing system. This system utilizes the mg2mcfg compiler described in Guillaumin [2004] as a front end. The parser features wild-card parsing over unknown segments of arbitrary unknown length, after Lang [1988] for use over probabilistic grammars as part of a psycholinguistic modelling tool for computing the entropies [Hale, 2006] and surprisals [Hale, 2001] of expressive grammars intersected with automata.

The document begins with theoretical background of the MCFG formalism, probabilistic prefix parsing, surprisal, and entropy in Section 1. In Section 2, we provide a brief user guide for the parser. In Section 3, we provide developer documentation on the main parsing algorithm and data structures. The document specifically focuses on the probabilistic components of parsing, including the renormalization method [Nederhof and Satta, 2006] for finding the true intersection PCFG conditioned on a prefix. We also examine how the entropy of this intersection grammar is computed, using matrix inversion.

1.1 Multiple Context Free Grammars

Multiple Context Free Grammar [H. Seki and Kasami, 1991] is a mildly-context sensitive Avarind K. Joshi and Weir. [1992] formalism, as are Minimalist Grammar [Stabler, 1997], Combinatory Categorical Grammar [Steedman, 2000], and Tree Adjoining Grammar [Joshi, 1985].

Multiple context free grammar rules (and more generally, mildly context sensitive grammar rules) differ from context-free rewriting rules in delineating abstract syntax from concrete syntax. Abstract syntax refers to our method of rewriting nonterminals as other non-terminals, whereas concrete syntax refers to how we managae the string yields of our symbols.

In a context-free production

$$\alpha \rightarrow \beta\gamma \tag{1}$$

α, β and γ represent strings, and the rewriting operation denoted by $\rightarrow$ conflates a category-forming operation and a concatenation operation. The category-forming operation and concatenation operation are disassociated in mildly context sensitive grammars since the basic unit is non-concatenate (think of a derived tree in TAG, which permits adjunction into it), and concrete syntax permits fancier string yield operations than simple concatenation.

Multiple Context Free Grammar rules operate over tuples of strings. MCFG rules are of the form

$$A_0 \rightarrow f[A_1, A_2, ...A_q] \tag{2}$$

where the function f takes as arguments tuples of strings and returns an A_0 which is also a tuple of strings.

MCFG rules are often notated in practice with the following notation of Dan Albro [Albro], which makes clear the evaluation of f.

$$t7 \rightarrow t4\ t5\ t6\ [(2,0)][(1,0);(0,0);(1,1)] \tag{3}$$

This example can be read off as follows. An index (x, y) refers to the yth member (in 0-initial list notation) of the x argument. The semicolons represent concatenations, whereas the brackets

designate members of the yield tuple. Thus, in the above example, a new tuple of category t7 is formed, where the first member is simply the 0th member of the $t6$ tuple, and the second member is formed from the concatenation of: the 0th member of $t5$; the 0th member of $t4$; and the 1st member of $t5$. A rule such as the above example, where the second member of $t7$ is formed from the interpolation and subsequent concatenation of members of $t4$ and $t5$, can thus be seen as similar to a movement operation (MG), wrap rule (CCG), or adjunction instance (TAG).

We could simulate CFG in MCFG by introducing trivial string yield functions which just always concatenate, always producing categories whose string yields are always 1-tuples.

$$S \rightarrow NPVP[(0,0);(1,0)] \tag{4}$$

We can do this because CFG is equivalent to what is called 1-MCFG. If the rewriting rules of CFG yield only 1-tuples [M=1], then 2-MCFG will allow 2-tuples (we will say, have FAN-OUT of 2). MG are equivalent to 2-MCFG (their derivation tree languages are the same). The movements in MG are represented as string yield functions which range over the components of the categories, so that Merge instance will translate into a binary MCFG rule, and Move instance translate into unary MCFG rules.

We require our MCFG to be what H. Seki and Kasami [1991] term linear and non-erasing. Linearity restricts possible string yield functions to those which have range of type tuple of string. It thus bans nesting of tuples inside other tuples in string yield functions. H. Seki and Kasami [1991] show that such nesting would extend the power of MCFG to Turing-completeness. The non-erasingness condition requires that for any MCFG rule, the string yield function cannot 'disappear' any of the components of the children's string yields. H. Seki and Kasami [1991] show that erasing MCFG are not a strict extension of non-erasing MCFG, and that any erasing MCFG can be rewritten to an equivalent non-erasing MCFG.

1.2 MCFG String Parsing

We present a sample MCFG derivation. In 1.2, we present a small MCFG which captures noun phrase movement for unaccusative verbs such as 'fell'.

S	$\rightarrow$	VP	$[(0,1);(0,0)]$
VP	$\rightarrow$	V NP	$[(0,0)][(1,0)]$
NP	$\rightarrow$	D N	$[(0,0);(1,0)]$
V	$\rightarrow$	'fell'	
N	$\rightarrow$	'boy'	
D	$\rightarrow$	'the'	

Figure 1: Sample MCFG

The VP rule in this MCFG implements the desired mapping between 'deep structure' and 'surface structure' by seperating 'abstract' and 'concrete' syntax. The VP rule's abstract syntax creates a VP symbol by taking the NP as the 'complement' of V, but the string yield function over VP evaluates as ("fell","the boy"). The S rule is therefore able to 'move' this NP up in its concrete syntax, as seen in the sample derivation in 2.

In our sample derivation we included backpointers in the Parsing as Deduction style [Shieber et al., 1995]; we can retrieve a derivation by following backpointers into subderivations. Importantly,

9	S	$[Frag_{(0,3)}]$	$\rightarrow$ 8
8	VP	$[Frag_{(0,2)}, Frag(2,3)]$	$\rightarrow$ 6,7
7	NP	$[Frag_{(0,2)}]$	$\rightarrow$ 4,5
6	V	$[Frag_{(2,3)}]$	$\rightarrow$ 3
5	N	$[Frag_{(1,2)}]$	$\rightarrow$ 2
4	D	$[Frag_{(0,1)}]$	$\rightarrow$ 1
3	'fell'	$[Frag_{(2,3)}]$	$\rightarrow$ Term
2	'boy'	$[Frag_{(1,2)}]$	$\rightarrow$ Term
1	'the'	$[Frag_{(0,1)}]$	$\rightarrow$ Term

Figure 2: Sample MCFG Derivation

the derivation is just a grammar [Billot and Lang, 1989] conditioned on the full string; this fact will enable us to port analytical tools from traditional PCFGs over incremental parse states.

1.3 Prefix Parsing as Intersection of (M)CFG and Finite State Automaton

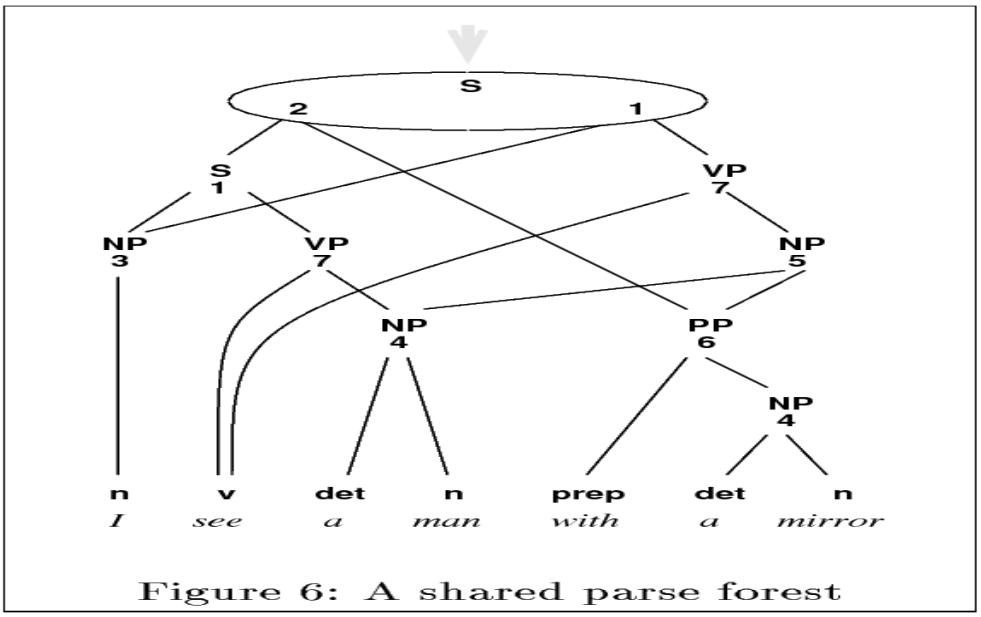

Figure 3: Billot and Lang [1989], Lang [1988]: Ambigious Parse, as Shared Parse Graph

1.4 Probabilistic Parsing and MCFG

PCFGs have the following properties [Manning et al., 1999]

- Place Invariance: The probability of a subtree does not depend on where in the string the words it dominates are.
- Context Free: The probability of a subtree does not depend on words not dominated by the subtree.

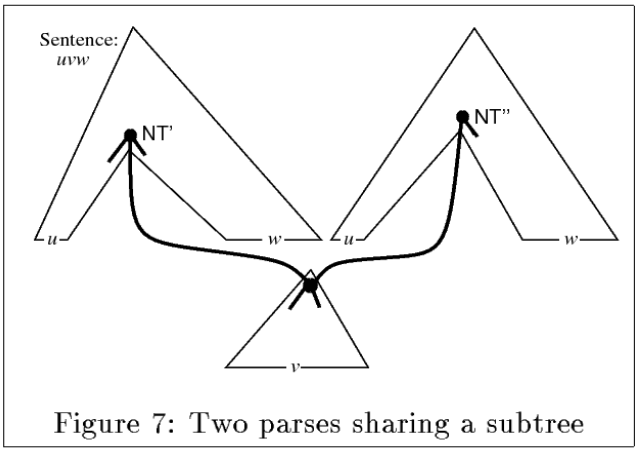

Figure 7: Two parses sharing a subtree

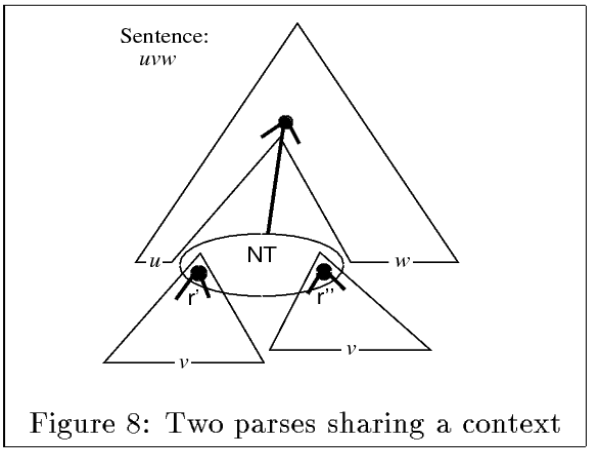

Figure 8: Two parses sharing a context

Figure 4: Billot and Lang [1989], Lang [1988]: Ambigious Parse, as Item/Tree Sharing

9	S	$[Frag_{(0,2)}]$	$\rightarrow$ 8
8	VP	$[Frag_{(0,2)}, Wild(2)]$	$\rightarrow$ 6,7
7	NP	$[Frag_{(0,2)}]$	$\rightarrow$ 4,5
6	V	$[Wild_{(2)}]$	$\rightarrow$ 3
5	N	$[Frag_{(1,2)}]$	$\rightarrow$ 2
4	D	$[Frag_{(0,1)}]$	$\rightarrow$ 1
3	'*'	$[Wild_2]$	$\rightarrow$ Term
2	'boy'	$[Frag_{(1,2)}]$	$\rightarrow$ Term
1	'the'	$[Frag_{(0,1)}]$	$\rightarrow$ Term

- Ancestor Free: The probability of a subtree does not depend on nodes in the derivation outside the subtree.

Our MCFG derivation trees have a context free 'abstract syntax' [H. Seki and Kasami, 1991, Kallmeyer, 2010]. The MCFG abstract syntax in fact enjoys Place Invariance, Context Freeness, and Ancestor Freeness via its "context-free backbone" [Avarind K. Joshi and Weir., 1992], permitting the extension of PCFG methods to more expressive mildly context sensitive grammars. The parameters for productions in a probabilistic mildly context sensitive grammar can thus be estimated via (Weighted) Relative Frequency Estimation for PCFG [Chi, 1999], as shown in .

$$P(A \rightarrow \xi) = \frac{\sum_{i=1}^{n} f(A \rightarrow \xi; \tau_I)}{\sum_{i=1}^{n} f(A; \tau_I)} \tag{5}$$

Relative Frequency Estimation estimates the likelihood of an outcome by taking a count of the number of outcomes in a data set, and dividing by a count for all contexts in which the event could have had that outcome. For a PCFG G, the event of rewriting a parent A will have outcomes which are rules in G with left hand side A. To estimate the probability of a rule $A- > \xi$ from corpus τ, we only need count the number of occurences of $A \rightarrow \xi$ in τ as the numerator, and divide by the total number of instances of A as left hand side of a rule in τ.

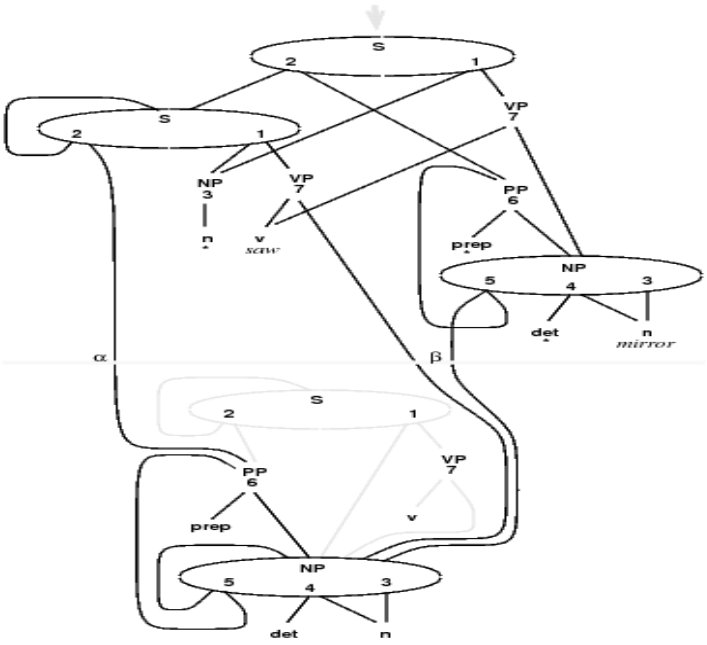

Figure 9: Full parse forest for an incomplete sentence

Figure 5: Intersections of Context Free Grammars and Automata

A PCFG with arbitrary probabilities on productions may fail to define a probabilistic language. For a PCFG G to define a probabilistic language, we require that P be *proper* and *consistent*[1]. Following Chi [1999], a PCFG is *proper* iff $\sum_{\lambda:A\rightarrow\xi)\in G} P(A \rightarrow \lambda) = 1$, i.e. for any nonterminal A, the probabilities on rules rewriting A sum to 1. A PCFG G is *consistent* if $\sum_{x\in\Sigma^*} P(S \Rightarrow x) = 1$, if the set of all strings derived by G have probabilities summing to 1.

That G is proper is not sufficient to ensure that G defines a probabilistic language. Stolcke [1995] demonstrates a PCFG which is proper but fails to define a probabilistic language, as shown in 1.4.

$$\begin{array}{lll} 2/3 & \text{S} & \rightarrow \text{S S} \\ 1/3 & \text{S} & \rightarrow \text{'a'} \end{array}$$

Figure 6: Inconsistent PCFG

Stolcke [1995] shows that the set of all sentences defined by this probabilistic grammar have probabilities which sum to greater than 1.

for all nonterminals X, the rule probabilities sum to 1

[1]Following here the terminology of Stolcke [1995],Chi [1999], Hale [2006], and not the terminology of Jelinek and Lafferty [1991], who uses *consistent* to term what we mean by *proper*.

consistent if all string probabilities add to 1.0. = if the PCFG's fertility matrix [2] A's spectral radius (largest eigenvalue) ≤ 1.0 = if the fertility matrix A is invertible.

Chi [1999] shows that if a PCFG is trained from a treebank with (Weighted) Relative Frequency Estimation, it is guaranteed to be consistent. The fertility matrix A is indexed by nonterminals, in which the (i, j) entry in A records the expected number of the nonterminal j from one rewriting of the nonterminal i. Grenander [1967], Stolcke [1995] show that the inversion $(I - A)^{-1}$ gives the transitive closure of the fertility relation; this is crucial for the computation of PCFG entropy and requires a consistent PCFG.

The spectral radius of A shows how recursive the PCFG is; if $i(A) \leq 1.0$, then a derivation will halt with certainty. If $i(A) \geq 1.0$, then non-terminals are being introduced 'faster' than they are rewriting into terminals, with the disastrous result that some strings gain infinite probability.

1.5 Probabilistic Intersection of a (M)CFG and Finite State Automaton

1.5.1 Inside Probability

We will need (for several reasons) to compute the inside probability of a category over string or automata. Following Manning et al. [1999, 392], we define the inside probability (β_N) of a nonterminal N on an automaton w given a grammar G by 6.

$$\beta_N(p, q) = P(w_{pq}|Npq, G) \tag{6}$$

Let $w_{(i,j)}$ be an automaton transition sequence between i and j consuming the symbol w. The inside probability $\beta_N(p, q)$ is the probability that a subtree rooted in N has the string yield $w_{(p,q)}$. We typically use dynamic programming via the inside algorithm to compute inside probabilities recursively. For a given nonterminal A, a PCFG G and right hand side ξ, let $P(A \to \xi|G)$ be the probability of the rule $A \to \xi$ according to G. The inside probability of a nonterminal A is defined inductively:

Base Case: For preterminal nodes A in G, $\beta_A(p, q) = P(A \to w_{p,q})|G)$.

Inductive Case: for other nonterminal nodes A in G, for binary rules of the form $A \to B_1C_1$, $A \to B_2C_2...$, $\beta_A = \sum_{R(A)\in G} \beta(B_i)\beta(C_i)$.

1.5.2 Renormalization by Inside Probability

Nederhof and Satta [2008] describe the computation of weighted intersection PCFG. Their method obtains the renormalized probability of a situated rule (which we situate wtih indices with (x, y),(x_1, y_1)...) as a product of the original probability of the unsituated rule and the inside probabilies of situated categories, according to:

$$P'(A_{(x,y)}) \to B_{(x_1,y_1)}) = \frac{r(A \to B)\beta_{B(x_1,y_1)}}{\beta_{A_{(x,y)}}} \tag{7}$$

$$P'(A_{(x,y)} \to B_{(x_1,y_1)}C_{(x_2,y_2)}) = \frac{r(A \to BC)\beta_{B_{(x_1,y_1)}}\beta_{C_{(x_2,y_2)}}}{\beta_{A_{(x,y)}}} \tag{8}$$

This requires determining the inside probability of a rule, as described in O'Donnell et al. [2009]. Renormalization by inside probability of the branch reflects the true information that an incremental prefix parse contains. To condition a probabilistic context free grammar G on a finite

[2] equivalently, momentum matrix, first-moment matrix

state automata w, we need a weighted intersection G' whose categories are intersections of categories in G with state transitions in w. The probabilities on productions in G' need to somehow reflect both probabilities of rules in G and probabilities of state transitions in w.

The inside probability of a category C over a string yield w is the conditional probability of C deriving w given the grammar G and initial probabilities of productions in G. β_C moves closer to 0.0 as $w|G$ becomes less likely. Take an example where we have a grammar G with two rules, $PA \rightarrow B_1$, and $P_1A \rightarrow B_2$, where P and P_1 are the initial probabilities from training. To determine renormalized probabilities P' and P'_1 on the weighted intersection of G and w, P' should be lowered relative to R' to the extent that $\beta_{B_1} \leq \beta_{B_2}$. Since the initial probabilities P and P_1 are fixed from training, the inside probabilities reflect estimated 'counts' of rules from w, which drives home the intuitive simularity between the renormalization equations in 7 and 8 and the equation for Relative Frequency Estimation, presented again in 9.

$$P(A \rightarrow \xi) = \frac{\sum_{i=1}^{n} f(A \rightarrow \xi; \tau_I)}{\sum_{i=1}^{n} f(A; \tau_I)} \tag{9}$$

This approach is distinct from pooling probability from rules that do not appear in the intersection directly over to rules that do. Given the previous example, we might set P equal to $P + P_1 = 1$ when $A \rightarrow B_2$ fails to appear in the intersection of G and w. Naive renormalization of this type can fail to propogate the information of a string event up through the grammar. When we renormalize by inside probability, we adjust the probabilities of rules whenever the inside probabilities of children non-terminals change. Intuitively, the probability of a rule is reduced whenever any path between the rule and the string yield are eliminated; inside probability propagates this information up to nonterminals higher in the parse. However, naive renormalization reduces the probability of a rule R only when all paths from R are falsified. Naive renormalization overweights many branches because it can adjust probabilities of rules low in the tree without adjusting probabilities of ancestor branches.

For example, renormalization by the inside probability of the branch insures that branches with zero probability and branches with low probability are treated proportionally. Naive renormalization does not insure this because rules only lose probability when they fail to show up in a parse tree, and not when they show up with low probability. An unguarded naive-renormalized derivation admits items to the chart even if those items have zero probability. This could occur, for instance, in the following pathological case, where two rules ($D \rightarrow F$,$E \rightarrow GH$) are admitted to the training grammar even though they did not show up in treebank.

The unguarded parse of the string '1 2 3' incorrectly gives probability to the branch headed by A-¿D E even though it should have zero probability.

This occurs because when we parse non-probabilistically, we admit zero probability branches into the parse; when we perform naive renormalization, we eliminate non-present items low in the tree, moving the probability of these eliminated rules to their siblings, but we do not do so high in the tree, because those categories are present in the parse as well.

If we parse with naive renormalization and 'with guards', by only admitting items to the chart when they have non-zero probability, we sidestep this problem for now.

We obtain an identical answer if we parse and then renormalize by inside probability.

Now imagine a less pathological case. The two zero count rules in 7 now have one attestation each in treebank; we adjust the sibling rule counts accordingly.

1.0	100 S	→	A
0.5	50 A	→	B C
0.5	50 A	→	D E
1.0	50 B	→	F
1.0	50 C	→	G H
0.0	0 D	→	F
1.0	50 D	→	X
0.0	0 E	→	G H
1.0	50 E	→	Y
1.0	50 F	→	'1'
1.0	50 G	→	'2'
1.0	50 H	→	'3'
1.0	50 X	→	'9'
1.0	50 Y	→	'8'

Figure 7: Pathological Training Example

1.0	S	→	A
.5	A	→	B C
.5	A	→	D E
1.0	B	→	F
1.0	C	→	G H
0.0	D	→	F
0.0	E	→	G H

Figure 8: Pathological Training Example, Naive Renormalization Without Guards

We again parse the string '1 2 3'. Naive renormalization without guards still incorrectly leaves the rules with parent A equiprobable.

However, naive renormalization with guards still has problems; it does not adjust the probabilities of rules rewriting A because all children of A are attested in the chart, even though some children have very low probabilities.

Renormalizing by inside probability of the branch sidesteps this issue because it always reallocates probabilities whenever any path from non-terminal to child has been falsified.

Thus, renormalization by inside probability of the branch insures that whether zero-probability items are filtered out of the chart or not, we will obtain the same result. Naive renormalization, however, cannot deliver on this guarantee. The difference between 8 and 9 is exactly the problem the parser faces on whether or not to enter zero probability items into the chart. Mcfgcky can emulate naive renormalization, so we tested whether the policy towards parsing non-zero items affected either of naive or inside renormalization. Parsing the prefix "the butter melted *" on unerguacc5.mcfg, we obtained the following entropies:

1.6 Entropy

We can conceptualize entropy in terms of a discrete random variable. The entropy of a discrete random variable is equal to $-\Sigma_i p_i \log_2 p_i$ index. A fair coin, for example, has an entropy of $-(.5$

1.0 S → A
1.0 A → B C
0.0 A → D E
1.0 B → F
1.0 C → G H
0.0 D → F
0.0 E → G H

Figure 9: Pathological Training Example, Naive Renormalization With Guards

1.0 S → A
1.0 A → B C
0.0 A → D E
1.0 B → F
1.0 C → G H
0.0 D → F
0.0 E → G H

Figure 10: Pathological Training Example, Renormalization by Inside Probability

$log\ .5) + (.5\ log\ .5)$, i.e., 1.0 bits of entropy, since each of the two possible outcomes (heads, tails) has a 0.5 probability of occurring.

Grenander [1967] demonstrates the computation of entropies of Probabilistic Context Free Grammars. For a PCFG with production rules in Chomsky normal form, let the set of production rules in G be Π, and for a given nonterminal ξ denote the set of rules with parent ξ as $\Pi(\xi)$). The entropy associated with a single rewrite of ξ is given by Equation 10

$$H(\xi) = -\sum r \in \Pi(\xi_i) p_r r \log_2 p_r \tag{10}$$

A PCFG is a random process whose outcome is a derivation, and the PCFG's total entropy is the entropy associated with derivations in Π, where each derivation is a series of rule selection events. Then the entropy of a PCFG is equal to the total entropy of the start symbol S, where the entropy associated with one-step rewrites of ξ must inherit entropy associated with rewriting children of rules in $\Pi(\xi)$.

Grenander [1967]'s Theorem in Equation 11 provides a recurrence relation for determining the entropy of the start category S; each parent accrues entropy from children weighted by the probabilities of those children.

$$H(\xi_i) = h(\xi_i) + \sum r \in \Pi(\xi_i) p_r [H(\xi_{j1}) + H(\xi_{j2}) + ...] \tag{11}$$

The theorem also provides a closed-form solution when the probabilistic context free grammar is recursive or otherwise impractical to compute. The closed-form solution uses linear algebra to efficiently compute the entropy of the hierarchical process in two parts: the 'local' entropies of parents as simple random variables, and a fertility relation. Let $\vec{h}$ be a vector indexed by nonterminal symbols with each component given by Equation **??**.

1.0	100	S	→	A
0.5	50	A	→	B C
0.5	50	A	→	D E
1.0	50	B	→	F
1.0	50	C	→	G H
0.02	1	D	→	F
0.98	49	D	→	X
0.02	1	E	→	G H
0.98	49	E	→	Y
1.0	50	F	→	'1'
1.0	51	G	→	'2'
1.0	51	H	→	'3'
1.0	49	X	→	'9'
1.0	49	Y	→	'8'

Figure 11: Less Pathological Example

1.0	S	→	A
.5	A	→	B C
.5	A	→	D E
1.0	B	→	F
1.0	C	→	G H
1.0	D	→	F
1.0	E	→	G H

Figure 12: Possible Training Example, Naive Renormalization without Guards

$$h_i = h(\xi_1) = -\sum_{r \in \Pi(\xi_i)} p_r r \log_2 p_r \tag{12}$$

Record the one-step fertility relation in a matrix A, labelled with non-terminals, where $A_{i,j}$ is the expected number of j is the number of i to appear in one rewriting of i. Then the vector of total entropies associated with non-terminals in G is given by Equation 13.

$$H_G = (I - A)^{-1}\vec{h} \tag{13}$$

For example, the local entropy of a category C according to a pcfg G is the entropy of a die whose sides are labeled and weighted according to one-step rewritings of C. The inversion $(I-A)^{-1}$ gives the transitive closure of the fertility relation: the expected number of j in a derivation issuing by any number of steps from i. The dot product of right hand side vector $\vec{h}$ and $(I-A)^{-1}$ gives a vector of total entropies for each non-terminal, including S.

2 Quick User Guide to mcfgcky

Mcfgcky is built to work with the command line, and it's various functions are triggered by distinct 'modes'. Each mode has its own syntax, and the command line parsing at present is rigid as to this syntax.

1.0 S → A
.5 A → B C
.5 A → D E
1.0 B → F
1.0 C → G H
1.0 D → F
1.0 E → G H

Figure 13: Pathological Training Example, Naive Renormalization with Guards

1.0 S → A
0.996 A → B C
0.004 A → D E
1.0 B → F
1.0 C → G H
1.0 D → F
1.0 E → G H

Figure 14: Possible Training Example, Renormalization by Inside Probability

	Without Guards	**With Guards**
Naive	4.517 b.	4.431 b.
Inside	4.885 b.	4.885 b.

2.1 Debugging Mode

"Usage: mcfgcky -d grammar-file 3.sentence >\n"

Given an mcfg file at location 'grammar-file' and a sentence set in double quotes, attempt to recognize the sentence.

EXAMPLES:

./mcfgcky -d grammars/parsingtest/bever.mcfg "the horse raced past the barn *"

Recognize the Kleene closure of the prefix 'the horse raced past the barn' according to bever.mcfg

./mcfgcky -d grammars/parsingtest/bever.mcfg "the horse raced past the barn fell"

Recognize the sentence 'the horse raced past the barn' according to bever.mcfg

2.2 Parsing Mode

"Usage: mcfgcky -p grammar-file 3.dict-file ¡4.-s 5.sentence/4.-f 5.in-file¿ ¡6.l/-pp¿ 7.¡mcfg/-mg/-mgd¿ / 6.-dcfg¿ 7-8.outfile \n"

Given an mcfg file at location 'grammar-file' and a sentence set in double quotes, or a corpus of sentences, attempt to parse.Return a sample of derivation trees in mcfg or mg output. MG output requires a dictionary file from the Guillamin compiler –for mcfg output, this file is never checked.

Options: -s/-f: if -s, parse the sentence given in double quotes. If -f, parse the unweighted corpus of sentences in location 'in-file'. Each sentence in this corpus is double-quoted, each is delimited with semicolon and newline. The formatting on this corpus is sensitive: please leave at least one line of whitespace at the end, and prepare the corpus in a bona-fide text editor, as some Mac (TextEdit) and Windows text editors will leave carriage returns which confuse filereader.ml, which parses the corpus into a useful format.

Example: “the horse raced past the barn”; “the dish ran away with the spoon”;

-l/-pp: if -l, latex-output for qtree. if -pp, relatively horrible text output.

-dcfg: output a product-sum graph of the situated context-free grammar, for dot. Requires an outfile name.

-mcfg/-mg: if -mcfg, mcfg derivation tree output. if -mg, mg derivation tree output. -mgd will produce an X-bar style MG-derived tree, not yet implemented.

EXAMPLES:

: ./mcfgcky -p grammars/unergunacc/unergunacc4.mcfg grammars/unergunacc4.dict -s ”the horse raced past the barn *” -l -mg unergunacc.out*

Parse the Kleene closure of the prefix ’the horse raced past the barn’ according to unergunacc4.mcfg, returning a sample of mg derivation trees in qtree latex format.

: ./mcfgcky -p grammars/unergunacc/unergunacc4.mcfg grammars/bever.dict -f unerg.test.txt -l -mcfg unergunacc.out*

Parse the corpus located at unerg.test.txt, return mcfg trees in latex, requires a dummy dictionary file which will not be read.

: ./mcfgcky -p grammars/unergunacc/unergunacc4.mcfg grammars/unergunacc4.dict -s ”the horse raced past *” -dcfg unergunacc.out*

Parse the Kleene closure of the prefix ’the horse raced past’ according to unergunacc4.mcfg, and return the and-or graph of the grammar conditioned on this prefix.

2.3 Statistical Mode

”Usage: mcfgcky -i/-ii 3.trainingcorpus/pcfg 4.test-file -l/-q 5.out-file\n”ˆ

Generate or read in a probabilistic model for cfg or mcfg, and compute prefix probabilities, surprisals, and entropies for each prefix.

Options: -i/-ii: if -i, induct a P(M)CFG from a weighted corpus by building a mini-treebank and using Weighted Relative Frequency Estimation.

In addition to the above cautions regarding corpora, this corpus should be formated as follows:

(0.379, “the horse raced past the barn”);

(284e-6,“the dish ran away with the spoon”);

Corpreader.ml should have no problem with floats in standard or scientific notation to any reasonable number of places.

if -ii, read in a pcfg. (Train.ml, Pcfgreader.ml)

This PCFG should be formatted as follows.

“0.00064636 SUBJP → PRON NBAR”

“0.39936363 SUBJP → PRON”

Leave a line of whitespace at the end. At present, this mode is redundant in that it must read in the pcfg and the derivative mcfg seperately. Great confusion can result if the two do not match up; the typical method as of late has been to develop the pcfg and then derive the trivial cfg. Also, not much is known as how this mode works on non-context free grammars...there should be no problem, but it has not really been testsed.

TESTING

3 Design

Mcfgcky is a bottom-up chart parser in the style of Shieber et al. [1995]. As such, it approaches parsing as a deduction problem, where the axia are the string terminals and a set of inference rules specifies the formulation of derived categories. Both the axia and derived items are contained

within a chart which records the entirety of the bottom-up deduction and thus eliminates any need for backtracking.

3.1 Chart Items

As such, the basic data structure in Mcfgcky is the chart item, which represents some hypothesis made by the recognizer for the category corresponding to a certain string segment. In Mcfgcky, this structure is of the OCaML type *item*, such as follows.

- type item = {prob:float;pointer:int;cat:string;pos:span list}

The items populating the chart of a MCFG recognizer in the Shieber et al. [1995] style, like that of a CFG CKY recognizer, must minimally contain a category label (cat) and a mapping from that category to the input string (pos). In a CKY recognizer, this representation of the input string segment often takes the form of a tuple of integers (x, y), where x and y denote the positions which begin and end the string. This convention succinctly and uniquely identifies the string segment which the parser has attached a category label, and determines which items are concatenable. As the yield function of an MCFG rules takes not strings but tuples of strings as arguments, and returns such, an item's mapping of categories to the input string is more complex. In Mcfgcky, the field *pos* achieves this more complex mapping as a list of *spans*, where a span represents some maximal contiguous item contained within an MCFG tuple.

The parametrized type *span* implements the range of automata we wish to parse.

The Empty and Var types represent categories and portions of categories that are not directly instantiated by the input string. In the automaton, these types implement epsilon transitions which generate a symbol without consuming an input symbol. The distinction between Empty and Var in the MCFG parser repects a distinction made by the Guillamin compiler from MG. A span with empty string yield could correspond to an MG category which has no string yield, such as a functional head. It could also correspond to an ad hoc category fashioned by the compiler to fascilitate such rules as head movement and adjunction. In the former case, we term the empty span Empty and in the latter case, we term it Var.

The type Wild is a constructor of an index (implemented as integer), reflecting a self-loop in the automaton that consumes any symbol in the input while producing a fixed symbol in the output. The type Frag is a constructor of a tuple of indicies, representing a transition from one index to another in the automaton producing a symbol. The abstraction of a tuple of spans is implemented in Mcfgcky as a list of spans, as OCaML permits homogeneous lists of arbitrary length.

Overall, the record item type maps a single MCFG category to potentially more than one segment of the input string, and permits the arguments of MCFG rewrite rules to be non-concatenate.

The field *point* operationalizes the backpointing system required to fashion a parser from a recognizer. Throughout parsing, a backpointing hash table is populated in tandem with the chart, and *point* simply identifies the key in this table which corresponds to this item. At the end of parsing, the pointers for items which meet the success criterion are read off, and serve as the starting points for the backpointing traversal function, $checkmcfg$ or $checkmg$, which yields derivation trees corresponding to the MCFG or MG derivation tree, respectively.

Finally, the field *prob* represents the probability of the item given the input string, which serves probabilistic parsing.

3.2 Inference Rules

Mcfgcky can handle grammars whose rules are unary or binary. As a result of this, Mcfgcky employs three main inference rules: **unaryf**, which is unary; **left** and **right**, which are binary. **left** and **right** refers simply to which member of the binary rule is in the trigger and which is in the chart, and do not refer to relative positions in the string. These rules all take as their arity an agenda **og**, a grammar **g**, the pointer hashtable **phash**, a trigger item **trigger**, and the chart **chart**.

```
let left og g phash trigger chart =
    for comp=0 to n in chart
    for rule=0 to n in
    List.filter (rule's first rhs argument = trigger.cat) grammar
    if List.length (righthandside(rule)) =2 and
    trigger.cat = (righthandsides(rule).[0]) and
    comp.cat = (righthandsside(rule).[1]]
    then f(rule,trigger,comp)::og
    else og
```

left is seen in the above pseudocode to map a given trigger item to all members of the of the chart and with all rules in the grammar containing the category of the trigger item as first argument (second argument in right, only argument in unaryf). **left**,**right**,**unary** simply match premises to pertinent axiom schemata, returning (rule, trigger, comp) triples (or a (rule, trigger) tuple in the unary case) where the categories of trigger and comp match the right hand side of rule in the correct order. The final inference is implemented separately as the evaluation of the yield function f on the (rule, trigger, comp) triple, by the *populate*, *populate*2, and *concat* functions. *populate*2 simply builds a new item given the (rule, trigger, comp) triple, in the binary case, and *populate* does likewise in the unary case. *populate*2 and *populate* both include sanity checks, and both use the *concat* function to determine the *pos* field of the new item.

As the pos field of the item is a span list, *concat* simply takes as arguments lists of spans (or span list list) and returns their concatenation, a span list; e.g. concat [[Fragment(2,3);Fragment(3,4); Fragment(4,5)];[Fragment(5,6)]] will return the concatenate [Fragment (2,5);Fragment(5,6)], while concat [Fragment(2,3);Wild(4,5);Fragment(5,7)] will return [Fragment(2,3);Fragment(4,7)]. *spanlistlist* is given a type definition as an *anonitem*; as OCaML lists are homogenous and of arbitrary length, the use of the outer list and the temporary data structure enables f to take as input an arbitrary number of arguments (or equivalently, an n-tuple of arbitrary length)

Recall that a Multiple Context Free Grammar rule has arguments and yield that are tuples of strings. Whereas the yield function of a context-free grammar rewriting rule conflates category rewriting and string concatenation, the yield function of a multiple context free grammar combines the category rewriting and string concatenation operations in a more complex mapping. Revisiting again the example of a context free grammar rewriting rule...

$$\alpha \rightarrow \beta\gamma \tag{14}$$

In a context free grammar, β must map onto a portion of the string which is prior to (left of) the part of the string which γ maps onto. This is not so in a Multiple Context Free Grammar.

$$A_0 \rightarrow f[A_1, A_2, ...A_q] \tag{15}$$

$$t7 \rightarrow t4\ t5\ t6\ [(2,0)][(1,0);(0,0);(1,1)] \tag{16}$$

The yield function f is free to interpolate and concatenate the members of its argument tuples independently. It follows that the MCFG parser must delineate the concatenation function from the inference rules somewhat, as a given inference rule may not exhibit any instance of concatenation.

3.3 Agenda and Exhauastion Mechanism

3.4 Backpointing

Mcfgcky uses a separate chart to record derivational information needed to print analyses. As each items is inserted into the main chart, a key,value pair is recorded in the backpointer chart, and the backpointer key recorded in the main chart item's backpointer field.

When the exhaustion mechanism closes the chart, and items matching the success criterion are retrieved, the pointers can be read off the success items and seed the backpointing mechanism.

3.5 Output

In parsing modes, Mcfgcky can output a variety of derivation tree outputs in LaTeX, plain text, or dot. It can print out either MCFG derivation trees or MG derivation trees, provided the corresponding dictionary file from the Guillaumin [2004] is provided for the reverse compilation of MCFG categories back into MG feature prefixes. It does not at present output MG derived trees. Mcfgcky also will print out a product-sum graph in both parsing and statistical modes.

In statistical modes, Mcfgcky prints out reports with pertinent information. In the verbose statistical mode, Mcfgcky prints out the treebank grammar, derivation grammars, inside probabilities, $\vec{h}$, and fertility matrices, and entropies for each prefix. *vech* and the fertility matrix could probably be shunted off into a debugging mode. In the quiet statistical mode, Mcfgcky prints out surprisals, entropies, and inside probability of the start symbol (prefix probabilities) for each prefix.

3.6 Training

3.7 Testing

Nederhof and Satta [2008] show that the number of iterations to convergence is dependent on the spectral radius. Convergence can be out of reach in pathological cases with $sr \approx 0.\bar{9}$, but convergence is readily obtainable for $sr \leq 0.7$, which generally holds on natural language grammars.

Setting $n = 100$ should be sufficient for almost all natural language grammars trained from corpora, but caution is warranted when grammars have large spectral radii. We ran a series of tests on gazdar.mcfg, which is a quite loopy CFG with a spectral radius of INSERT. 15 show the results of parses of three and four word prefixes being parsed as n is incremented. The X-axes depict n, with entropies on the Y-axises. Entropy scores converged out to 16 significant digits around 50 n.

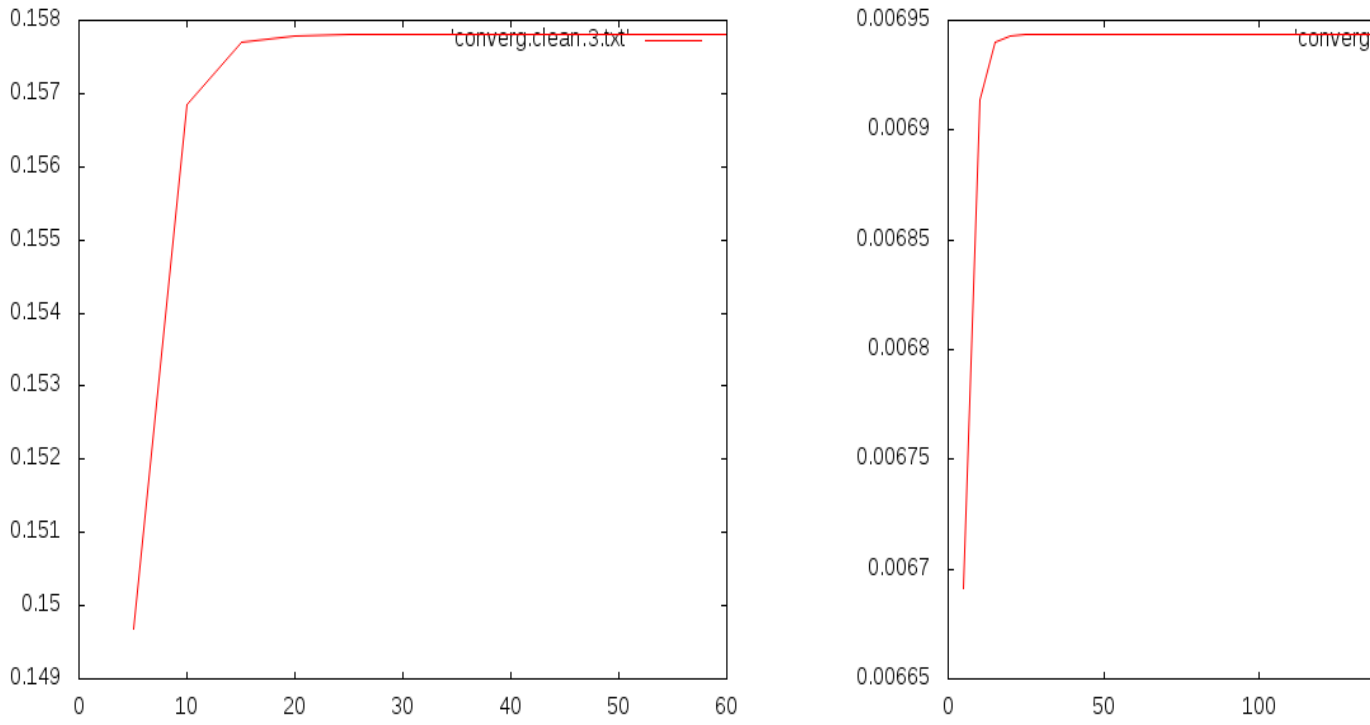

Figure 15: Convergence of gazdar.cfg on three and four word prefixes

3.8 Surprisal

3.9 Entropy

4 Implementation

Myqueue implements a functional queue structure used in the agenda mechanism, and Seq (as coded in Paulson [1996] implements lazy lists used in point.ml to retrieve individual trees from the forest. Utilities provides data manipulation utilities. Print and Output contain the string-manipulation and output utilities used in printing different outputs to the toplevel, to the standard out, and to file.

Item contains the *item* record type, which uses the string *cat* type for category labels, the *span* type for mapping to the input string, and the integer *point* type for the backpointing mechanism. Concat implements the computation of string yield functions, which perform concatenations and range restrictions, and also contains several boolean check functions which ensure that derived items are interpretable.

Grammar contains the types related to the mcfg grammar data structure. Mcfgread and Mcfglex are a lexer/parser combination used for reading in this grammar object from an mcfg file, which is output by the guillamin compiler.

Rules implements the inference rule schemeta which are used by the parsing engine, while Mcfgcky contains the parsing engine. Point implements the packed shared forest. Decompile handles reverse compilation from MCFG to MG if desired. Dictlex and Dictread provide a lexer/parser used to read in a dictionary file provided by the Guillamin compiler. This dictionary file consists of

tuples of MCFG categories and their MG interpretations, and is used by Mcfgcky to set the success criterion and to provide a translation of the MCFG parse into MG.

Fileread/Filelex,Corpread/Corplex, and Pcfgread/Pcfglex all read in unweighted corpora, weighted corpora, and pcfg for statical mode parsing. Fileread/filelex is also used via batch processing in parsing mode. Pcfg implements basic structures for probabilistic parsing at training time and test-time. Train trains up a model from a corpus using Weighted Relative Frequency Estimation over a treebank generated by the parser over the corpus from the grammar. *Pcfglex/Pcfgread* read in an arbitrary PCFG directly. Test implements the testtime functionality of computing intersection grammars and associated data which are prerequisite for entropy computations. Dcfggraph uses ocamlgraph to topologically sort the grammar into buckets. Inside computes the inside probability of catgeories and calculates renormalized probabilities of the intersection grammars. Entropy uses Grenander's Theorem and camlgsl to invert a fertility matrix and obtain entropies over probabilistic grammars.

Print is responsible for printing to standard out and providing string translations of data structures, while Output assembles reports that are output to a desired file. Dcfgdisplay uses dcfggraph to output product-sum graphs if desired.

4.1 Utilities, Basic Data Structures, and Input Modules

4.1.1 Module `Utilities`

Basic utilities for mcfgcky.

4.1.2 Module `MyQueue`

4.1.3 Module `Seq`

4.1.4 Module `Mcfgread`

Read in a Multiple Context Free Grammar in Albro notation.

4.1.5 Module `Fileread`

Read in an unweighed corpus in basic list format.

4.1.6 Module `Corpread`

Read in an weighed corpus.

4.1.7 Module `Pcfgead`

Read in an arbitrary model in probabilistic context free grammar format.

4.2 Parsing Modules

4.2.1 Module `Item`

An item in our chart is a tuple of a category and its range-restricted string yield. The string yield is tuple-valued, where each component could either correspond to an actual string, an empty string, or Kleene star.

```
type cat = string
```

Categories are simple strings.

```
type index = int
```

We represent an item's position in the string using an index of type int.

```
type span =
  | Var
  | Emp
  | Wild of index
  | Fragment of index * index
  | Hyp of index
  | Fail
```

A span is a component of the tuple-valued range-restriction of the string yield. Each span represents an automaton transition, so that the total string yield represents the sequence of automaton transitions that are necessary to derive the decorated item. Vars and Emps represent epsilon transitions in our system; they are distinguished in MCFG parsing because they have unique status in the MG items that gave rise to them. Vars represent the empty string yields of functional head categories in the source MG, whereas Emptys represent the empty string yields of ad-hoc categories introduced solely by the Guillaumin [2004] compiler.

Wilds represent Kleene closures in string yields, and serve to reflect that a given category may be present in the right context of a prefix. A Fragment (x,y) represents an automaton transition from index x to index y where $y > x$. We represent a range-restriction as a Fragment when that span's derivation maps on to at least one instance of a real string. Thus, our string concatenation rules define an algebra (a semiring) over tuples of strings. Loosely, we realize (+) as string concatenation, and (*) as a function over tuples of strings yielding tuples of strings. The concatenation of a Fragment (x,y) and a Wild (y) is Fragment (x,y), where Wild(y) is a zero-item for concatenation. A (linear, non-erasing) string yield function for a unary rules which takes the 0th member of a child's string yield is the null-item for (*).

```
type item = {
  cat : cat ;
  pos : span list ;
}
```

An item is a category situated over (range-restricted by) an n-tuple of spans in the string. This homogenous n-tuple of spans is implemented as a list of spans. An item is implemented as a record with category (cat) and string yield (pos) fields.

4.2.2 Module `Grammar`

Grammar.ml implements the data types and functions used in Multiple Context Free Grammar. Whereas a Context Free Grammar production conflates a symbol-forming operation and a concatenation operation, MCFG divorces abstract and concrete syntax by allowing arbitrary string yield functions with more powerful operations than concatenation. The domain and range of (Linear) MCFG string yield functions is defined over yields of type tuple of strings; this is guranteed by the linearity of (L)MCFG without which, MCFG would be Turing-equivalent.

```
type daughterID = int
```

```
type member_of_tupleID = int
type componentSpec = daughterID * member_of_tupleID
type rewriter =
  | Func of componentSpec list list
  | Terminal
```

We implement our string yield functions according to the Albro notation. For a rule $A \rightarrow BC[(0,0);(1,0)][(0,1)]$, we evaluate the string yield of A as a tuple with two components (where each component is indicated in brackets), where the first component is the concatenation of respective 0th members of string yields of B and C, and the second component is the 1th component of the 0th category, B. When A is a preterminal, we indicate as such with the category Terminal. daughter_ID indicates which category is referred to; member_of_tupleID indicates which component of some category's string yield is referred to; such that componentSpec can uniquely indicate a span in B or C. Rewriter implements the tuple-valued rewrite function as a list of componentSpec.

```
type mcfgrule = Item.cat list * rewriter
```

An mcfgrule has abstract syntax (of type 'cat list', where we always take the 0th member of the list to refer to the parent, and the other members of the list to refer to the children) and concrete syntax (of type rewriter).

```
type pmcfg = mcfgrule list
```

A (Parallel) Multiple Context Free Grammar is a list of mcfgrules.

```
val tupler : 'a * rewriter -> rewriter
```

Given a mcfgrule, obtain the string yield function.

```
val cats_of_rule : 'a * 'b -> 'a
```

Given a mcfgrule, obtain the abstract syntax as a list of categories.

```
val lhs : 'a list * 'b -> 'a
```

Given a mcfgrule, obtain the parent category (left hand side) of the rule.

```
val rhs : 'a list * 'b -> 'a list
```

Given a mcfgrule, obtain the children categories (right hand side) of the rule.

```
type opt_mcfg = {
  parent_ord : (Item.cat, mcfgrule) Hashtbl.t ;
  lchild_ord : (Item.cat, mcfgrule) Hashtbl.t ;
  rchild_ord : (Item.cat, mcfgrule) Hashtbl.t ;
  onchild_ord : (Item.cat, mcfgrule) Hashtbl.t ;
  ispreterm_ord : (bool, mcfgrule) Hashtbl.t ;
}
```

Implement an optimized MCFG object where any quantity of interest asked for by rules.ml is indexed at training time and retirevable in constant time at testing time. Another optimization would be to index by linearization arity as well; this would be relatively easy to implement, but may not be of great use: our MCFG categories generally have constant lineaarization arity.

```
val orderbyparent :
  ('a list * 'b) list -> ('a, 'a list * 'b) Hashtbl.t -> unit
```

Populate a hash table where each rule is indexed by parent.

```
val orderbylchild :
  ('a list * 'b) list -> ('a, 'a list * 'b) Hashtbl.t -> unit
```

Populate a hash table where each rule is binary and indexed by the left child.

```
val orderbyrchild :
  ('a list * 'b) list -> ('a, 'a list * 'b) Hashtbl.t -> unit
```

Populate a hash table where each rule is binary and indexed by the right child.

```
val orderbyonchild :
  ('a list * 'b) list -> ('a, 'a list * 'b) Hashtbl.t -> unit
```

Populate a hash table where each rule is unary and indexed by the only child.

```
val orderbypreterm :
  (string list * rewriter) list ->
  (bool, string list * rewriter) Hashtbl.t -> unit
```

Populate a hash table where each rule is a preterminal production, indexed by the preterminal.

```
exception WeirdRule of mcfgrule
```

Exception for abberant mcfg rules.

```
class grammar : pmcfg ->
  object
```

```
    method get_rules_for : Item.cat -> Grammar.mcfgrule list
```

Get any rules where category is a parent.

```
    method get_rules_left_child : Item.cat -> Grammar.mcfgrule list
```

Get any binary rules where category is a left child.

```
    method get_rules_right_child : Item.cat -> Grammar.mcfgrule list
```

Get any binary rules where category is a right child.

```
    method get_rules_only_child : Item.cat -> Grammar.mcfgrule list
```

Get any unary rules where category is a child.

```
    method get_all_empty_preterm : Item.cat list
```

Get any preterminals whose string yield is empty.

```
    method get_all_nonempty_preterm : Item.cat list
```

Get any preterminals whose string yield is non-empty.

```
    method get_all_empty : Item.cat list
```

Get any preterminals whose string yield is empty (deprecated).

```
    method get_all_head : Item.cat list
```

Get any preterminals whose string yield is non-empty (deprecated).

```
    method get_all_cats : Item.cat list
```

Get all categories in the grammar.

```
    method get_start_symbol : string list
```

```
  end
```

Get the start symbol of the grammar.

```
val opt_gram : Grammar.opt_mcfg
```

The optimized grammar object.

```
val unopt_gram : Grammar.pmcfg
```
Return the list-formatted unoptimized grammar.

```
method get_all_rules : Grammar.pmcfg
```
Get all the rules in the grammar.

4.2.3 Module `Point`

Implements the backpointer system and subsumption check, which we use to construct a packed parse forest.

```
type point =
  | Term
  | Unary of Item.item * Grammar.componentSpec list list
  | Binary of Item.item * Item.item * Grammar.componentSpec list list
```

Like in CFG, a pointer gives us the abstract syntax used to enter some category, by referring to either one or two parents or a terminal. Unlike CFG, it should also indicate the concrete syntax used to form it, by including the string yield function which was used.

```
type sitrule = Item.item * point
```
Situated rules are non-terminals decorated or range restricted with their string yields. This could be cleaned up by having one parameterized type, which is either Situated of Item.item * point, or Nonsituated of Item.cat * catlist.

```
type sitcfg = sitrule list
type dcfgrhs =
  | Bin of Item.cat * Item.cat
  | Un of Item.cat
  | T
```

A Dcfg gives us unsituated rules back from the situated parse forest, for when we need parse trees to train rules with unsituated categories. This works effectively, but is conceptually clunky.

```
type dcfgrule = Item.cat * dcfgrhs
type dcfg = dcfgrule list
exception StrangeChart
```
Exception for when our chart is strange.

```
val crossproduct : 'a list -> 'b list -> ('a * 'b) list
```
Cross product of two lists.

```
val string_of_span : Item.span -> string
val sitcfgOfChart : (Item.item, point) Hashtbl.t -> Item.item -> sitcfg
```

Obtain the dcfg from a chart given the pointer hash table and the current parent/key to work on.

`val wlength : string -> int`
The length of the sentence that the situated start symbol needs to span over.

`val sit_start_symbol : Item.index -> Item.item`
Given wlength, return the situated start symbol.

`val gr_sort : (Item.item * 'a) list -> (Item.item * 'a) list`

Guarantee that items are sorted longest to shortest, with start symbol first. Not strictly necessary, but should speed processing.

```
val sortedcfgOfChart :
  (Item.item, point) Hashtbl.t -> Item.item -> sitrule list
```

Sorted pcfg from chart according to the above.

`val desitCFG : (Item.item * point) list -> (Item.cat * dcfgrhs) list`

For a situated context free grammar, return the desituated dcfg version

`val catlist_of_dcfg : ('a * dcfgrhs) list -> 'a list`

`val catarray_of_dcfg : ('a * dcfgrhs) list -> 'a array`
Return the list of categories in a dcfg. Used to index the momentum matrix.

`val abstract : Item.item -> Item.cat`
Desituate a situated category.

`val abstract_rule : point -> dcfgrhs`
Desituate a situated rule.

`val range : int -> int -> int list`

A range operator, which I'm sure is implemented elsewhere

`val success_items : string -> Item.item list`

Given a string, return a list of successful situated start symbols according to the approp string length

```
class forest :
  object
```

`val point_hash : (Item.item, Point.point) Hashtbl.t Pervasives.ref`

A hashtable of pointers which provides the main functionality of the packed parse forest. In trunk, this is a standalone hash table, not bundled into the forest object.

`val subsumptionok : 'a -> ('a, 'b) Hashtbl.t -> bool`

Boolean check for whether parent item is already in chart; if it is, include the new pointer but don't add the item again.

`val (@@) : 'a Seq.seq -> 'a Seq.seq -> 'a Seq.seq`

`val (@::) : 'a -> 'a Seq.seq -> 'a Seq.seq`

`val lcrossproduct : 'a Seq.seq -> 'b Seq.seq -> ('a * 'b) Seq.seq`

```
val treesOfChart :
  (Item.item, point) Hashtbl.t ->
  Item.item -> Item.item Utilities.stufftree list
```

Operators for lazily extracting a sample of parse trees out of the forest. Could be improved.

4.2.3.1 Experimental Kilbury Parsing

```
val cathash : (Item.cat, Item.item) Hashtbl.t Pervasives.ref
val chart : Item.item list Pervasives.ref
val grams_of_chart : (Item.item, Point.sitcfg) Hashtbl.t
method cats : (Item.cat, Item.item) Hashtbl.t
method pointers : (Item.item, Point.point) Hashtbl.t
method get_sit_cats : Item.cat -> Item.item list
method subsume_ok : Item.item -> bool
method add_pointer : Item.item -> Point.point -> unit
method add_cat : Item.item -> unit
method add_to_chart : Item.item -> Item.item list
method get_chart : Item.item list
method incr_chart : unit
method incr_pointers : unit
method incr_cats : unit
method hash_grams : string -> unit
method sitgram : Item.item -> Point.sitcfg
end
```

4.2.4 Module `Concat`

Concat.ml implements concrete syntax via string yield functions for MCFG.

```
type anonitem = Item.span list list
```

An anonitem is an intermediate representation of the string yield function.

4.2.4.1 Integrity Checks

```
val reasonable_ok : Item.span -> bool
```

Check that indices on a given range restriction component (span) are sensible. Frag(x,y) requires $0 \leq x \leq y$. Wild(x) requires $0 \leq x$.

```
val concatenable_ok : Item.span list -> bool
```

Check that a given instance of concatenation is sensible. A concatnenation $[(x, y); (x_1, y_1 ...]$ requires that $y = x_1$, etc.

`val overlap : Item.span -> Item.span -> bool`

`val overlap_ok : Item.span list -> bool`

Overlap_ok applies overlap to deliver T iff candidate range restrictions don't have overlapping spans, F o/w. A range restricted string yield $(Frag(x,y), Frag(x_1,y_1))$ requires that $\neg\exists z.(x \leq z \leq y) \wedge (x_1 \leq z \leq y_1)$

4.2.4.2 Sanity Concatenation Checks

`val wholerule_ok : 'a list list -> anonitem -> bool`

T iff ensure that entire mcfg rule is used in creating item, and rule was used non-erasingly, F o/w.

`val concat_fail : Item.span list -> bool`

Checks to see if item contains the Fail itemlet, created by bad, incomplete concatenation.

`exception ConcatFail of string`

Exception thrown if concatenation fails, to be caught safely by rules.ml which will not populate the chart in this instance.

4.2.4.3 Concatenation Functions

`val concat : Item.span list -> Item.span`

; is the concatenation function, defined on range restrictions. Concatenation is not a total function, as things ot be concatenated need to be adjacent (for $[Frag(x,y); Frag(x_1,y_1]$, concatenation is valued only when $y = x_1$). Given a span list, concatenates into a single span. Nonsense concatenations should return the Fail span type, propogating information up to the sanity check

Emp and Var are 'free' in the string and place no restrictions on where in the string they can be concatenated in; thus they are cached out at concatenation.

`exception GrabNth of string`

`val grab : Grammar.componentSpec -> anonitem -> Item.span`

Given an anonitem, returns the nth member of the nth tuple. Throws GrabNth is member is out of range of the arity of the tuple.

`val mapconcat : Grammar.componentSpec list -> anonitem -> Item.span`

Map concatenation across all the gets in a single spec list– eg `(0,1);(1,0)`. Each instance of mapconcat builds one component of the parent's string yield tuple.

`val f : Grammar.componentSpec list list -> anonitem -> Item.span list option`

The yield function maps mapconcat across the anonitem with the whole rewrite–eg `(0,1);(1,0)0,0`.

4.2.5 Module `Rules`

Rules.ml defines the inference rules we use to populate our chart, faciliating Shieber et al. [1995]-style parsing as deduction. This modularity would allow relatively easy adaption of all the statistical stuff to say, Earley parsing. It also handles the actual population of the chart.

```
val unfunc : Grammar.rewriter -> Grammar.componentSpec list list
```

Unbox the rewriter type. Doing this sparingly instead of at runtime every time a string yield function is called actually yields a signifcant performance gain.

4.2.5.1 Chart Population Functions

```
val populate2 :
  Item.cat list * Grammar.rewriter ->
  Item.item ->
  Item.item ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  'b list
```

Populate2 factors out population, item checking, and string yield evaluation from the left and right inference rules. It applies the 'concrete syntax' by calling concat.ml with the specified yield function. It checks as well that the string yield function is sensible before sending it off to concat.ml.

```
val populate :
  Item.cat list * Grammar.rewriter ->
  Item.item ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  'b list
```

Populate2 factors out population, item checking,a nd string yield evaluation from the unary inference rules. It applies the 'concrete syntax' by calling concat.ml with the specified yield function. It checks as well that the string yield function is sensible before sending it off to concat.ml.

```
val is_nonzero_prob :
  Item.cat list * 'a -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> bool
```

Boolean check to ensure that items at training time have nonzero probability. This is well-tested, and does not effect accuracy at the expense of performance so long as inside renormalization is used.

```
val non_erasing_bin : 'a list -> 'b list -> Grammar.rewriter -> bool
```

A cheap filter to reduce calls to concat.ml, which implements a cheap approximate and conservative (false negatives but not false positives) check that the rule is non-erasing. Checks that comps in x + comps in y = linearzation arity of rule r.

```
val non_erasing_un : 'a list -> Grammar.rewriter -> bool
```

A cheap filter to reduce calls to concat.ml, which implements a cheap approximate and conservative (false negatives but not false positives) check that the rule is non-erasing. Checks that #comps in x + #comps in y = linearzation arity of rule r.

4.2.5.2 Inference Rules

```
val left :
  < get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    get_sit_cats : Item.cat -> Item.item list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'c -> 'b list
```

Given the left member of a rhs binary rule, map over all pertinent rules and all pertinent items with the correct right member.

```
val right :
  < get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    get_sit_cats : Item.cat -> Item.item list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'c -> 'b list
```

Given the right member of a rhs binary rule, map over all pertinent rules and all pertinent items with the correct left member

```
val unaryf :
  < get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'c -> 'b list
```

Given the only child of a rule, map over all pertinent unary rules with this member.

4.2.5.3 Ground Axia/Initial Chart Population

```
val empties :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list; .. > ->
  'a -> < add_pointer : Item.item -> Point.point -> 'b; .. > -> Item.item list
```

Populate all empties.

```
val can_hypo : string -> bool
```

Populate the chart based on any ellipsis cues (so far these are indicated with).

```
val items_of_just_string :
  < get_all_nonempty_preterm : Item.cat list;
    get_rules_only_child : string -> (Item.cat list * 'a) list; .. > ->
  string ->
  < add_pointer : Item.item -> Point.point -> 'b; .. > -> Item.item list
```

Populate all nonterminals for the string.

4.2.5.4 Kilbury Style Inference Rules

```
val items_of_token :
  < get_all_nonempty_preterm : Item.cat list;
    get_rules_only_child : string -> (Item.cat list * 'a) list; .. > ->
  string ->
  < add_pointer : Item.item -> Point.point -> 'b; .. > ->
  Item.index -> Item.item list
```

Inference rules for kilbury. Experimental and not accurate.

```
val populate2_up :
  Item.cat list * Grammar.rewriter ->
  Item.item ->
  Item.item ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  'c -> 'b list
```

Binary chart population for kilbury. Experimental and not accurate.

```
val populate_up :
  Item.cat list * Grammar.rewriter ->
  Item.item ->
  < add_pointer : Item.item -> Point.point -> 'a;
    add_to_chart : Item.item -> 'b list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  'c -> 'b list
```

Unary chart population for kilbury. Experimental and not accurate.

```
val left_up :
  < get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  'a ->
  < add_pointer : Item.item -> Point.point -> 'b;
    add_to_chart : Item.item -> 'c list;
    get_sit_cats : Item.cat -> Item.item list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'd -> 'c list
```

Binary inference rule for kilbury. Experimental and not accurate.

```
val right_up :
  < get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  'a ->
  < add_pointer : Item.item -> Point.point -> 'b;
    add_to_chart : Item.item -> 'c list;
    get_sit_cats : Item.cat -> Item.item list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'd -> 'c list
```

Binary inference rule for kilbury. Experimental and not accurate.

```
val unaryf_up :
  < get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  'a ->
  < add_pointer : Item.item -> Point.point -> 'b;
    add_to_chart : Item.item -> 'c list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'd -> 'c list
```

Unary inference rule for Kilbury. Experimental and not accurate.

```
val delid : Item.item * Item.item -> bool
```

Experimental Deletion under Identity Check for Conjunction Parsing. Untested.

```
val noncyclic : Item.span list * Item.span list -> bool
```

Experimental Cyclicity Check for Conjunction Parsing. Untested.

```
val is_hyp : Item.item -> bool
```

Experimental Hypothesis Check for Conjunction Parsing. Untested.

```
val rel_hyp : ('a, Item.item) Hashtbl.t -> 'a -> Item.item list
val left_discharge :
  'a ->
  'b ->
  < add_pointer : Item.item -> Point.point -> 'c;
    cats : (Item.cat, Item.item) Hashtbl.t;
    points : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'd -> Item.item list
```

Experimental Disharge Rule for Hypothesis/Conjunction Parsing. Untested.

```
val right_discharge :
  'a ->
  'b ->
  < add_pointer : Item.item -> Point.point -> 'c;
    cats : (Item.cat, Item.item) Hashtbl.t;
    points : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. > ->
  Item.item -> 'd -> Item.item list
```

Experimental Disharge Rule for Hypothesis/Conjunction Parsing. Untested.

```
val items_of_string :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_only_child : string -> (Item.cat list * 'a) list; .. > ->
  string ->
  < add_pointer : Item.item -> Point.point -> 'b; .. > -> Item.item list
```

4.2.6 Module `Mcfgcky`

The main parsing module of mcfgcky.

```
val distribute :
  (Item.item -> 'a -> 'b list) list ->
  Item.item -> < get_chart : 'a; .. > -> 'b list
```

Distributes a trigger throughout the chart.

```
val distribute_debug :
  ('a list -> 'b) ->
  (Item.item -> 'c -> 'a list) list ->
  Item.item -> < get_chart : 'c; .. > -> 'a list
```

Distributes a trigger throughout the chart with debugging output.

```
val exhaust :
  'a ->
  < add_cat : Item.item -> 'b; get_chart : 'c; subsume_ok : Item.item -> bool;
    .. > ->
  Item.item Myqueue.BatchedQueue.queue ->
  (Item.item -> 'c -> Item.item list) list -> 'c
```

Exhaustion mechanism, proceeds until the chart is empty, which to the extent the algorithm is valid and complete will gives us the closure of the ground items under the inference rules.

```
val exhaust_debug :
  'a ->
  < add_cat : Item.item -> 'b; get_chart : 'c; subsume_ok : Item.item -> bool;
    .. > ->
  (Item.item list -> 'd) ->
  Item.item Myqueue.BatchedQueue.queue ->
  (Item.item -> 'c -> Item.item list) list -> 'c
```

Exhaustion mechanism with debugging output.

```
val parse_debug :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  string ->
  (< add_cat : Item.item -> 'b; add_pointer : Item.item -> Point.point -> 'c;
     add_to_chart : Item.item -> Item.item list; get_chart : 'd;
     get_sit_cats : Item.cat -> Item.item list;
     pointers : (Item.item, Point.point) Hashtbl.t;
     subsume_ok : Item.item -> bool; .. >
   as 'a) ->
  'a
```

Given a grammar g, sentence string w, and point side-table, run the parsing engine without backpointer retrieval and with debugging statements.

```
val parse :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_start_symbol : 'a; .. > ->
  string ->
  (< add_cat : Item.item -> 'c; add_pointer : Item.item -> Point.point -> 'd;
```

```
    add_to_chart : Item.item -> Item.item list; get_chart : 'e;
    get_sit_cats : Item.cat -> Item.item list;
    pointers : (Item.item, Point.point) Hashtbl.t;
    subsume_ok : Item.item -> bool; .. >
  as 'b) ->
 (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> 'b
```

Given a grammar g, sentence string w, and point side-table, parse with the appropriate decompilation of backpointers. Not used in statistical modes.

```
val parse_gen :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  string ->
  (< add_cat : Item.item -> 'b; add_pointer : Item.item -> Point.point -> 'c;
     add_to_chart : Item.item -> Item.item list; get_chart : 'd;
     get_sit_cats : Item.cat -> Item.item list;
     pointers : (Item.item, Point.point) Hashtbl.t;
     subsume_ok : Item.item -> bool; .. >
   as 'a) ->
  'e -> 'f -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> 'a
```

Given a grammar g, sentence string w, point side-table and a training pcfg, parse without appropriate decompilation of backpointers for use in statistical modes.

4.2.6.1 Experimental Kilbury Stuff

```
val all_points : ('a, 'b) Hashtbl.t -> ('a * 'b) list
```

```
val parse_up :
  < get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    .. > ->
  Item.cat ->
  (< add_cat : Item.item -> 'b; add_pointer : Item.item -> Point.point -> 'c;
     add_to_chart : Item.item -> Item.item list; get_chart : 'd;
     get_sit_cats : Item.cat -> Item.item list;
     pointers : (Item.item, Point.point) Hashtbl.t;
     subsume_ok : Item.item -> bool; .. >
   as 'a) ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> Item.index -> 'a
```

Experimental Kilbury stuff, doesn't work yet.

```
val parse_kilbury_gen :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_start_symbol : 'a; .. > ->
  string ->
  (< add_cat : Item.item -> 'c; add_pointer : Item.item -> Point.point -> 'd;
     add_to_chart : Item.item -> Item.item list; get_chart : 'e;
     get_sit_cats : Item.cat -> Item.item list; hash_grams : string -> 'f;
     incr_cats : 'g; incr_chart : 'h; incr_pointers : 'i;
     pointers : (Item.item, Point.point) Hashtbl.t;
     subsume_ok : Item.item -> bool; .. >
   as 'b) ->
  'j -> 'k -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> 'b
```

Experimental Kilbury stuff, doesn't work yet.

```
val parse_kilbury :
  < get_all_empty : Item.cat list; get_all_head : Item.cat list;
    get_all_nonempty_preterm : Item.cat list;
    get_rules_left_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_only_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_rules_right_child : Item.cat -> (Item.cat list * Grammar.rewriter) list;
    get_start_symbol : 'a; .. > ->
  string ->
  (< add_cat : Item.item -> 'c; add_pointer : Item.item -> Point.point -> 'd;
     add_to_chart : Item.item -> Item.item list; get_chart : 'e;
     get_sit_cats : Item.cat -> Item.item list; incr_cats : 'f;
     incr_chart : 'g; incr_pointers : 'h;
     pointers : (Item.item, Point.point) Hashtbl.t;
     subsume_ok : Item.item -> bool; .. >
   as 'b) ->
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> 'b
```

Experimental killbury stuff, doesn't work yet.

4.3 Statistical Parsing

4.3.1 Module `Pcfg`

Pcfg.ml implements basic types and functions for probabilistic parsing.

```
type fertility_matrix = Item.item array * float array array
```

A fertility matrix is an array_matrix of floats indexed by situated items, where the member of the matrix indexed by (x,y) indicates the expected number of y obtained in one rewriting of x, for a probabilistic grammar.

```
exception RowZero
```

Throw RowZero when a row of the fertility matrix is entirely zeroes.

```
type bucketed_key = {
  category : Item.item ;
  index : int ;
}
```

For inside probability functions, we have to divide the pcfg into 'buckets', where each 'bucket' is a strongly connected component of the pcfg's corresponding graph structure. A bucketed key tells us which bucket of the pcfg the item of interest is located in.

```
type iteration = int
```

For inside probability functions, we want to know which 'iteration' of the inside algorithm a value was evaluated at.

```
type bucketed_inside_hash = (bucketed_key, iteration * float) Hashtbl.t
```

Implements a memomoized data structure that permits us to record iterations of the loopy inside algorithm, and look them up in constant time when we need values from a particular iteration or bucket.

```
type pcfgrule = {
  rhs : Point.dcfgrhs ;
  mutable count : float ;
  mutable prob : float option ;
}
```

A rule type for training time. Works efficiently but is conceptually clunky, and could be combined with the test time probabilist ruletype Pcfg.renormed_probcfgrule, as a parameterized type.

```
type pcfg = (Item.cat, pcfgrule) Hashtbl.t
```

A PCFG is a hashtable of pcfgrules indexed by parent categories.

```
type renorm_cfgrule = {
  rrhs : Point.point ;
  mutable prior : float option ;
  mutable postr : float option ;
}
```

A rule type for testing time. Works efficiently but is conceptually clunky, and could be combined with the test time probabilist ruletype Pcfg.probcfgrule, as a parameterized type.

```
type renorm_cfg = (Item.item, renorm_cfgrule) Hashtbl.t
```

A renorm_cfg (sic) is a hashtable of renorm_cfgrules pcfgrules indexed by parent items, which are situated.

```
type t_bank = {
  test_sitcfg : Point.sitcfg ;
  mutable test_pcfg : renorm_cfg option ;
  mutable inside : bucketed_inside_hash option ;
  mutable fmatrix : fertility_matrix option ;
  mutable invert : fertility_matrix option ;
  mutable entropy : (Item.item, float) Hashtbl.t option ;
  mutable hvector : (Item.item, float) Hashtbl.t option ;
}
```

For a particular situated forest, keep around all of the pertinent data structures in a record.

```
type testbank = ((string * Item.item) * t_bank) list
```

List all the intersection probabilistic grammars and t_banks by prefix per sentence.

```
val catarray_of_pcfg : ('a, 'b) Hashtbl.t -> 'a array
```

Retrieve the array of categories in a probabilistic context free grammar.

```
val fmatrix_of_training_pcfg :
  (Item.cat, pcfgrule) Hashtbl.t -> Item.cat array * float array array
```

Obtain the fertility matrix of the training grammar, for eigenvalue checks and computing entropy of the training pcfg.

```
val fmatrix_of_pcfg :
  (Item.item, renorm_cfgrule) Hashtbl.t ->
  Item.item array * float array array
```

Obtain the fertility matrix of an intersection pcfg.

4.3.2 Module `Train`

Train up the corpus-minitreebank for -i mode. Also handles the reading in of pcfg, most of which is done in pcfglex.mll/pcfgread.mly

```
exception Logzero
```

Exception if log is taken of 0.0.

```
exception NegLog of float
```

Exception if log is taken of a negative number.

```
type s_bank = {
  weight : float ;
  training_sitcfg : Point.sitcfg ;
  mutable training_cfg : Point.dcfg option ;
}
```

Basic record type that bundles information we need for training on a particular sentence in a treebank.

```
type revprob_bankrule = {
  rcount : float ;
```

```
  rprob : float ;
}
type revprob_bankcfg = (Item.cat * Point.dcfgrhs, revprob_bankrule) Hashtbl.t option
```

Optimized PCFG type where a rule probability can be indexed by children for lookup in constant time. Used by inside.ml. Could be hidden behind a fancier pcfg object, but doesn't create much clunkiness in practice (inside.ml only sees this representation of the pcfg).

```
type train = {
  sbank : (string, s_bank) Hashtbl.t ;
  mutable by_parent : (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ;
  mutable by_rule : (Item.cat * Point.dcfgrhs, revprob_bankrule) Hashtbl.t option ;
}
```

Basic record type of a pcfg trained off of a single sentence. A PCFG for an entire treebank can be extracted from all of these sentences easily.

```
val read_pcfg : string -> train -> unit
```

Directly read in a training pcfg.

```
val parse_train :
  string ->
  ('a ->
   string ->
   (< pointers : (Item.item, Point.point) Hashtbl.t; .. > as 'b) ->
   'c -> ('d, 'e) Hashtbl.t -> 'f option -> 'g) ->
  'a -> 'b -> 'c -> 'h -> train -> string -> unit
```

Train up the corpus-minitreebank for -i mode. Requires an initialized parse forest (empty chart), an empty grammar object, and checking functions provided by decompile.ml to train on a weighted corpus. Uses simple weighted relative frequency estimation (Chi 1999).

```
val countpcfg_init :
  (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
  Point.dcfg -> float -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t option
```

Increments counts across the treebank on a treebank-level pcfg object from the sentence-level information.

```
val totalcatn : ('a, Pcfg.pcfgrule) Hashtbl.t -> 'a -> float
```

Sum across all trees in treebank to obtain a count for the number of rules with parent 'a, for the denominator in weighted relative frequency estimation.

```
val relcount : ('a, Pcfg.pcfgrule) Hashtbl.t -> 'a -> Point.dcfgrhs -> float
```

For a given rhs (point.dcfgrhs) and a given parent ('a), return the number of counts of that rule across the treebank.

```
val relfreq : ('a, Pcfg.pcfgrule) Hashtbl.t -> 'a -> Point.dcfgrhs -> float
```

Find the relative frequency of a rule in the grammar, according to the number of times the parent A ('a) $\rightarrow$ B (Point.dcfgrhs) found divided by the total count of rhs occurences of A.

```
val relprobpcfg_init : ('a, Pcfg.pcfgrule) Hashtbl.t -> unit
```

Populate the pcfg's hash table object with one relative-frequency estimated rule.

```
val pcfg_of_train_init : 'a -> train -> unit
```

Initialize the pcfg object by finding all relative frequencies of all rules. Desituate dcfgs, compile and prob treebank rules in one step.

```
val trainf :
  bool ->
  string ->
  'a ->
  'b ->
  ('c ->
   string ->
   (< pointers : (Item.item, Point.point) Hashtbl.t; .. > as 'd) ->
   'e -> ('f, 'g) Hashtbl.t -> 'h option -> 'i) ->
  'c -> 'd -> 'e -> 'j -> 'k -> string -> train -> unit
```

A higher order training function parameterized by arguments, to pass in to treebank class. The parse can be agnostic as to which way it was trained.

```
class treebank : (train -> 'a) -> 'b ->
  object
```

```
    val optbank : Train.train
```

Our main unoptimized treebank structure which is hidden behind the optimized treebank object.

```
    method prob_by_rule : Item.cat * Point.dcfgrhs -> float
```

```
    method has_par : Item.cat -> bool
```

Treebank object method which defines a relation which is T if parent is in the grammar, F o/w.

```
    method has_rule : Item.cat * Point.dcfgrhs -> bool
```

```
    method pcfg : (Item.cat, Pcfg.pcfgrule) Hashtbl.t option
```

```
  end
```

An optimized pcfg object.

```
type parsorted_drule = {
  pindex : int ;
  size : int ;
  pdefcat : Item.item ;
  ppar : Item.item ;
  prhs : Point.point ;
}
```

This is for indexing rules by their right hand side to be used by inside.ml, enabling these rules to be sorted into buckets.

```
type parsorted_dcfg = (Item.item, parsorted_drule) Hashtbl.t
```

The actual representation of the pcfg which inside.ml sees. This is necessitated partially by the function in ocamlgraph which does our topological sorting (scc_array), and partially the kind of iteration desired through the sort output, where we need an index and index item for each list.

```
val index_dhash_init :
  (Dcfggraph.D.vertex * Point.point) list ->
  (Dcfggraph.D.vertex, parsorted_drule) Hashtbl.t -> unit
```

Initialize the optimized pcfg representation which inside.ml sees.

4.3.3 Module `Inside`

Using the treebank grammar and the testtime interesection grammar, computes inside probabilities for each category in the intersection grammar to renormalize that grammar to find the weighted intersection. Compute inside probabilities of all categories in the intersection grammar, and then find the renormalized, situated probability of a rule by:

Because of recursion, a category can contribute inside probability to itself, so must find the limit of the inside probability for each non-terminal. Goodman [1999] shows us this is doable–sort the grammar into strongly connected components, the inside prob of a category C in a bucket B is the supremum of its value in the inside semiring over B. Then iterate through buckets in topological order, finding the limit of all values in the bucket before moving on.

STRATEGY:

Assuming you call with s each time, and you iterate progressively up., a memoized value is accessible if all of the following hold:

Assumption 1. Previous buckets are static. The grammar's graph structure can be sorted into its strongly connected components. Assumption 2: You can iterate the inside computation within a given bucket to reach the supremum, then proceed on to the next bucket.

Then we proceed on a memoized, loopy version of the inside algorithm, in which we work in topological order of the buckets. In general, we memoize the inside algorithm by working top down then bottom up. For a given category A in training pcfg R and with productions $A \to B_1C_1, A \to B_2C_2...$, $\beta(A)$ depends on probabilities in R and $\beta(B_1), \beta(C_1)....$ If we have no appropriate inside probability value for A in cache, we recur into children and determine the inside probability of children.

For the bucketed, iterative version of this algorithm that can treat looping buckets, we only need to check the value for the most recent iteration of some category, and then determine if that value is appropriate. If a category is its own child, the correct iteration to retrieve for calculating $A_{i,j}$ is $A_{i,j-1}$. If a category only depends on values outside its bucket, it necessarily depends on previous bucket, since we have topologically sorted the buckets and we are working through the buckets bottom-up. Thus, we can compute its inside probability 'statically', because the inside probabilities of those children aere guaranteed not to change, by Assumptions 1 and Assumptions 2.

If the inside probability of a category A depends on the inside probability of a child B in the same looping bucket i as it, then we iterate through the bucket because the value of $\beta(A)$ depends on $\beta(B)$ and vice versa. We proceed by first computing a static value for B which depends only on categories outside i, then iterate through i so that $\beta(A_{i,j})$ depends on $\beta(B_{i,j})$, and $\beta(B_{i,j+1})$ depends on $\beta(A_{i,j})$. This can happen when B is what we term the 'entry item' for i; if our grammar is topologically sortable, then each bucket has such an entry item. This is not guaranteed in general

(it is what Goodman [1999] terms linear solvability), but in practice our grammars are sortable this way. We then only need to have two kinds of dynamic inside probability computation; one for non-entry items, which require either values from the same bucket and same iteration, or static values from outside the bucket, and one for entry items, which require values from the same bucket and the previous iteration, or static values from outside the bucket. It is then only necessary to ever keep at most two iterations of a bucket around.

We can factor out the control flow of obtaining an appropriate inside value from the inside algorithm itself, and that is exactly what we do. The current design does not use a stopping criterion other than a set number of iterations, as a more sophisticated stopping criterion caused problems, and was eliminated.

`exception Logzero`
Exception for logarithms of zero.

`exception NegLog of float`
Exception for logarithms of negative number. Both of these exceptions are redundant on training.ml.

`val maxtup : (int * float) list -> int * float`
Used for finding the maximum iteration number in a list of buckets.

`exception FirstRecurrence`
Exception for when the value we need is the entry item.

`val abstract_cat : Item.item -> Item.cat`
Extract a category from an item, probably redundant on operations in dcfg.ml.

`val all_abstract : Item.item * Point.point -> Item.cat * Point.dcfgrhs`
Extract a rule from a situated rule, probably redundant on operations in dcfg.ml.

`exception SubNormal`
Exception which should raise if our values are every subnormal (if they ever lose precision). They never do. Take that log probs.

`exception ImpossProb of string`
Exception if our value is ever greater than 1.0, which because there is so much recursion, could result in a lot of chaos. This was caused by inprecise control flow in a previous version, but now never occurs.

`exception NonsenseRetrieval`
Value that you wanted from hash doesn't exists. Suggests incorrect control flow.

```
val inside_complexiter :
  Item.item ->
  (Pcfg.bucketed_key, int * float) Hashtbl.t ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  (Item.item, Train.parsorted_drule) Hashtbl.t -> int -> int -> unit
```

Iterates the loopy inside algorithm over a particular bucket, one iteration. Control flow which proceeds in topological order over the buckets is enforced elsewhere, as is the stoping criterion.

`val print_inside : (Pcfg.bucketed_key, int * float) Hashtbl.t -> unit`
Print some inside values.

```
val list_of_terms : ('a, Train.parsorted_drule) Hashtbl.t -> ('a * int) list
```
Obtain a list of terminals from the hashtable. Probably redundant.

```
val inside_complex_bucket :
  Item.item ->
  (Pcfg.bucketed_key, int * float) Hashtbl.t ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  (Item.item, Train.parsorted_drule) Hashtbl.t -> int -> int -> unit
```
Iterate as desired over a particular bucket.

```
val maxbucket : ('a, Train.parsorted_drule) Hashtbl.t -> int
```

Functions to get info about bucket structure

```
val bucket_indices : ('a, Train.parsorted_drule) Hashtbl.t -> int list
```
Returns a list of int*bool tuple, where int is the index and true indicates the item recurred in the original. Used to derive a list of bucket indices that actually exist.

```
val parents_of_buckets :
  ('a, Train.parsorted_drule) Hashtbl.t -> (int * Item.item) list
```
Return the parents of every bucket, where the parent is a 'start symbol' for that bucket. The particular identity of the start symbol doesn't matter so much, as we will immediately recur down to the entry item, which is readily identifiable at run time.

```
exception SafeNth of int
val safenth : 'a list -> int -> 'a
```
Exception for retrieving the nth item when n is larger than the list size, probably redundant.

```
val inside_complex :
  (Pcfg.bucketed_key, int * float) Hashtbl.t ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  'a -> (Item.item, Train.parsorted_drule) Hashtbl.t -> unit
```
Main algorithm that calcuates inside probability for all categories of a pcfg. Computes buckets in topological order, and iterates through each bucket n times. N ¿ 40 is generally sufficient for values to converge. Lack of underflow and some testing suggest that iteration values much greater than 40 do not pose an accuracy problem.

```
val renorm_init :
  (Item.item, Pcfg.renorm_cfgrule) Hashtbl.t ->
  (Pcfg.bucketed_key, 'a * float) Hashtbl.t ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  (Item.item, Train.parsorted_drule) Hashtbl.t -> unit
val inside_renorm :
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
```

```
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  (Dcfggraph.D.vertex * Point.point) list ->
  (Dcfggraph.D.vertex, Pcfg.renorm_cfgrule) Hashtbl.t option *
  (Pcfg.bucketed_key, int * float) Hashtbl.t option
```

Find the renormalized, situated probability of a rule by::

$$r'(A->B) = r(A->B) * \beta\_B/\beta\_A r'(A->BC) = r(A->BC) * \beta\_B * \beta\_C/\beta\_A \quad (17)$$

where $\beta_(A)$ indicates the supremum of the inside probability.

4.3.4 Module `Test`

Handles initalizing and renormalizing pcfg values across the testbank. Calculates certain values required for entropy computation, such as *vech* and the fertility matrix. Also handles the creation of prefix automata from strings we wish to parse.

`exception UnsampledRule`

Exception for when rule to calculate with was unseen at training time.

`val init_priors : Pcfg.t_bank -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t -> unit`

Initialize the weighted intersection with values form training time. The unsampled rule case shouldnt happen unless something is wrong–because test dodesn't parse zero prob items.

`val renorm : ('a, Pcfg.renorm_cfgrule) Hashtbl.t -> unit`

Higher order function which parametrizes the renormalization of pcfg, so that testing can be agnostic to renormalization. Renormalizes either naviely or according to inside probability.

```
val inside_renorm_init :
  Pcfg.t_bank ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  unit
```

Initialize the values that inside.ml depends on.

`val keylist_of_pcfg : ('a, 'b) Hashtbl.t -> 'a list`

Obtain nonterminal indiced for various things in test.ml, probably redunandant.

`val lg : float -> float`

Binary logarithm function which treats the lg of 0 as 0.

```
val retrieve_bucket_probs :
  ('a, Pcfg.renorm_cfgrule) Hashtbl.t -> 'a list -> ('a, float list) Hashtbl.t
val entropy_of_bucket : float list -> float
```

Compute the one step entropy for a non-terminal for use in $\vec{H}$.

```
val hvector_pcfg_init : Pcfg.t_bank -> unit
```
Computes the one step entropy for each non-terminal for use in $\vec{H}$.

```
val expect : 'a -> Item.item -> ('a, Pcfg.renorm_cfgrule) Hashtbl.t -> float
```
Compute the fertility matrix for the pcfg.

```
val testbank_init :
  Pcfg.t_bank ->
  < pcfg : (Item.cat, Pcfg.pcfgrule) Hashtbl.t option; .. > -> unit
```
Across the test corpus, calculate all the things needed for entropy computation, according to naive renormalization.

```
val inside_testbank_init :
  Pcfg.t_bank ->
  < has_rule : Item.cat * Point.dcfgrhs -> bool;
    pcfg : (Item.cat, Pcfg.pcfgrule) Hashtbl.t option;
    prob_by_rule : Item.cat * Point.dcfgrhs -> float; .. > ->
  unit
```
Across the test corpus, calculate all the things needed for entropy computation, according to inside renormalization.

```
val automatize : bool -> string list -> string list
```
If autbool, then construct automata for all possible prefixes of a sentence. Intelligently handle any stars in the input. Enforces a policy of parsing, in order, Kleene star (the unconditioned grammar), each prefix of the sentence including the prefix spanning the sentence, and then the sentence as a sentence. The just Kleene-star portion could be turned off when entropy of the unconditioned grammar would be hard to compute where the conditioned grammar might be easy, and we don't care about the information value of the first word.

```
val parse_testbank :
  ('a -> string -> Point.forest -> 'b -> ('c, 'd) Hashtbl.t -> 'e -> 'f) ->
  'a ->
  'g ->
  'b ->
  'h ->
  (< pcfg : 'e; .. > as 'i) ->
  (Pcfg.t_bank -> 'i -> 'j) ->
  string -> bool -> string -> ((string * Item.item) * Pcfg.t_bank) list
```
Higher order function which takes in several other higher order functions, including the renormalization function, either testbank_init or inside_testbank_init, the desired output functions, etc...and coordinates test time computation of values to hand off to entropy.ml.

```
val range : int -> int -> int list
```
A Range operator which could go in utilities, and is probably redundant.

```
val parse_testbank_kilbury :
  ('a ->
   string ->
   'b ->
   'c ->
   ('d, 'e) Hashtbl.t ->
   'f ->
   < pointers : (Item.item, Point.point) Hashtbl.t;
     sitgram : Item.item -> Point.sitcfg; .. >) ->
  'a ->
  'b ->
  'c ->
  'g ->
  (< pcfg : 'f; .. > as 'h) ->
  (Pcfg.t_bank -> 'h -> unit) ->
  string -> 'i -> string -> ((string * Item.item) * Pcfg.t_bank) list
```

Experimental incremental parsing on testbanks, doesn't work yet.

4.3.5 Module `Entropy`

Compute all the entropies.

```
exception NoOption
```

```
val optget : 'a option -> 'a
```

Unboxing option types for entropy purposes, probably redundant.

```
exception UnequalLengths of (int * int)
```

Exception is which is thrown when matrix and h-vector have different lengths (are differently indexed by unique sets of nonterminals).

4.3.5.1 Stolcke matrix cut trick

```
val zap : 'a array -> int -> 'a array
```

Delete the nth row in the array.

```
val zap_col : 'a array array -> int -> 'a array array
```

Delete the nth column in the matrix.

```
val ins : 'a array -> int -> 'a -> 'a array
```

Insert a row of all zeros into the nth position in the array.

```
val ins_col : 'a array array -> int -> 'a -> 'a array array
```

Insert a column of all zeros into the nth position in the matrix.

```
val reduce : 'a array * float array array -> 'a array * float array array
```

For any n, if both row n and column are all 0.0, eliminate row and column n and return them after inversion. Stolcke tells us this should work.

`val blowup :`
`'a array * float array array -> 'a array -> 'a array * float array array`

For any reduced invert matrix, put back in all the rows and columns of all zero that we took out of the preinverted matrix. Stolcke tells us that this should work.

`exception NonSquare of string`

Exception which is thrown when matrix is non-square.

`exception ZeroDeterminant`

Exception which is thrown when determinant of the matrix is equal to zero, aka the matrix is noninvertible.

`exception EmptyMatrix`

Exception which is thrown when matrix is entropy (probably because a parse was fail and not identified as such). It usually raises when something wrong happened at training. Before including this, parser would return a strange fragmentation error, which is supposed to be impossible for OCaML, according to Jon Harrop. Turns out it is, it was gsl (which is written in C) getting an empty matrix to invert.

`exception TrainingSpectralRadius of string`

Exception which is thrown when spectral radius of the matrix is greater than or equal to 1.0. This is bad, but should never happen when we traing the grammar from Treebank, according to Chi. It could potentially happen from arbitrary PCFG. It means that the set of sentences has probability greater than 1.0, ergo we fail to define a probabilistic language.

`exception SpectralRadius of float`

Exception which is thrown when spectral radius of the matrix is greater than or equal to 1.0. This is bad, but shouldn't happen so long as our intersection grammar is consistent, which is suggested by Nederhof and Satta. Our intersection grammar should be consistent when both our training grammar is consistent and our automaton is consistent. It could potentially happen from arbitrary PCFG. It means that the set of sentences has probability greater than 1.0, ergo we fail to define a probabilistic language.

`val eigenlist : Gsl_matrix.matrix -> float list`

Give me the eigenvalues of the matrix. Used for the spectral radius check, potentially very interesting for other things.

`val eigencheck : float -> bool`

`val infnorm : float array array -> float`

The infinity norm on matrices. Used to determine a condition number on matrice formed from inf

$_{n}ormM * inf$

$_{n}ormM - 1$.

`val cond_number : float array array -> float array array -> float`

Condition number of the matrix times the input error equals total error of the inverted matrix. The condition number tells us how 'lossy' matrix inversion is. Condition numbers range from 1 to infinity.

```
val entropy_of_matrix :
  'a array * float array array ->
  ('a, float) Hashtbl.t -> ('a, float) Hashtbl.t
```

Compute the entropy of a fertility matrix by calculating $(I - A)^{-1}\vec{h}$. We can be polymorphic, and pass in either original or renormalized matrices/hvectors. Original values are read off of a record, renormalized values off of the matrix-particular record entry.

```
exception Catlength of int
val entropy_of_testbank :
  string ->
  string ->
  bool ->
  bool ->
  (Pcfg.t_bank -> Train.treebank -> 'a) ->
  ('b ->
   string ->
   Point.forest ->
   'c -> ('d, 'e) Hashtbl.t -> (Item.cat, Pcfg.pcfgrule) Hashtbl.t option -> 'f) ->
  'b ->
  Point.forest ->
  'c ->
  'g ->
  (Train.treebank -> ((string * Item.item) * Pcfg.t_bank) list -> string -> 'h) ->
  string -> ((string * Item.item) * Pcfg.t_bank) list
```

Compute entropies for the entire testbank.

```
val entropy_of_testbank_kilbury :
  string ->
  string ->
  bool ->
  'a ->
  (Pcfg.t_bank -> Train.treebank -> unit) ->
  ('b ->
   string ->
   (< pointers : (Item.item, Point.point) Hashtbl.t; .. > as 'c) ->
   'd -> ('e, 'f) Hashtbl.t -> 'g option -> 'h) ->
  ('b ->
   string ->
   'c ->
   'd ->
   ('i, 'j) Hashtbl.t ->
```

```
   (Item.cat, Pcfg.pcfgrule) Hashtbl.t option ->
   < pointers : (Item.item, Point.point) Hashtbl.t;
     sitgram : Item.item -> Point.sitcfg; .. >) ->
  'b ->
  'c ->
  'd ->
  'k ->
  (Train.treebank -> ((string * Item.item) * Pcfg.t_bank) list -> string -> 'l) ->
  string -> ((string * Item.item) * Pcfg.t_bank) list
```

Experimental kilbury mode doesnt work yet.

4.4 Printing and Output

4.4.1 Module `Decompile`

Given an mcfg tree labeled with points and a dictionary file from the Guillaumin [2004] compiler, decompiles into an mg tree labeled with a (string,category) tuple. This can then be passed into a print function.

```
val label_of_tree : 'a Utilities.stufftree -> 'a
```

Extract a given category label from the tree.

```
val rewr_of_tree :
  Item.item Utilities.stufftree ->
  (Item.item list, Grammar.componentSpec list list) Hashtbl.t ->
  Grammar.componentSpec list list
```

Extract a given pointer from the tree.

```
val mcfg_to_mcfg :
  string ->
  'a ->
  'b -> Item.item Utilities.stufftree -> (string * Item.cat) Utilities.disptree
```

Provide an mcfg disptree for a given mcfg stufftree.

```
val mcfg_to_mg_ation :
  string ->
  (Item.cat, string) Hashtbl.t ->
  'a -> Item.item Utilities.stufftree -> (string * string) Utilities.disptree
```

Provide an MG derivation tree for a given mcfg tree.

```
val mcfg_to_mg_ived :
  string ->
  (Item.cat, string) Hashtbl.t ->
  (Item.item, Point.point) Hashtbl.t ->
  Item.item Utilities.stufftree -> (string * string) Utilities.disptree
```

Attempt to provide an MG derived tree for a given mcfg tree. Not implemented.

```
val successes : Item.item list -> string -> Item.cat list -> Item.item list
```

Return a list of successful start items spanning the string. Probably should go in point.ml if possible.

```
val check_mcfg :
  < get_chart : Item.item list; pointers : (Item.item, Point.point) Hashtbl.t;
    .. > ->
  string -> 'a -> Item.cat list -> (string * Item.cat) Utilities.disptree list
```

Higher order checking function for mcfg to pass into mcfgcky.ml and other modules requiring MCFG derivation tree output.

```
val check_mg :
  < get_chart : Item.item list; pointers : (Item.item, Point.point) Hashtbl.t;
    .. > ->
  string ->
  (Item.cat, string) Hashtbl.t ->
  Item.cat list -> (string * string) Utilities.disptree list
```

Higher order checking function for mcfg to pass into mcfgcky.ml and other modules requiring MG derivation tree output.

```
val check_mg_ived :
  < get_chart : Item.item list; pointers : (Item.item, Point.point) Hashtbl.t;
    .. > ->
  string ->
  (Item.cat, string) Hashtbl.t ->
  Item.cat list -> (string * string) Utilities.disptree list
```

Higher order checking function for mcfg to pass into mcfgcky.ml and other modules requiring MG derived tree output. Not implemented.

```
val check_null : 'a -> 'b -> 'c -> 'd -> 'e list
```

Vacuous higher order checking function for mcfg to pass into Mcfgcky.parse_gen where it is used for statistical parsing modules which do not require explicit discrete derivation or derived trees.

4.4.2 Module `Output`

Various output functions that we need. Coordinates output in a quiet and a verbose mode. Could probably stuff some verbose mode functionality into a separate debugging mode. Alternatively, could just have the command line option spit out what you need (include a seperate option for the fertility matrix, for example).

```
val round : float -> float
```

Round a floating point number intelligently to 3 places after the decimal. Used sparingly.

```
val rounded_string : float -> string
```

Provide a rounded string representation of the floating point number.

```
val pp_H_hash : Pcfg.t_bank -> Pervasives.out_channel -> unit
```
Output entropy values from the hash table for verbose output. Could in most cases be shut off.

```
val string_of_fmatrix :
  (Item.item array * float array array) option -> string
```

```
val pp_fmatrices : Pcfg.t_bank -> Pervasives.out_channel -> unit
```
Output the fertility matrix for verbose output. Could definitely be suppressed, but should be kept around for debugging purposes.

```
val string_of_trainrhs : Point.dcfgrhs -> Item.cat
```

```
val pp_trainpcfg :
  < pcfg : (string, Pcfg.pcfgrule) Hashtbl.t option; .. > ->
  Pervasives.out_channel -> unit
```
Output the treebank grammar in verbose mode. Very useful to have around.

```
val pp_testcfg : Pcfg.t_bank -> Pervasives.out_channel -> unit
```
Output the weighted intersection grammar in verbose mode. Very useful to have around.

```
val pp_renormedhvect : Pcfg.t_bank -> Pervasives.out_channel -> unit
```
Output the values of $\vec{h}$. Could be suppresed but useful for debugging.

```
val pp_inside : Pcfg.t_bank -> Pervasives.out_channel -> unit
```
Output the inside probabilities of different values in verbose mode. Mostly useful to have around.

```
val pp_testbank_report :
  ((string * 'a) * Pcfg.t_bank) list -> Pervasives.out_channel -> unit
```
Output all the testtime-pretinent values provided by the above functions in a report in verbose mode.

```
val pp_ent_report :
  < pcfg : (string, Pcfg.pcfgrule) Hashtbl.t option; .. > ->
  ((string * 'a) * Pcfg.t_bank) list -> string -> unit
```
Total output for verbose mode.

```
val pp_ent_sum : 'a -> ((string * 'b) * Pcfg.t_bank) list -> string -> unit
```
Summary of entropy values for quiet mode.

```
val pp_surp_sum :
  'a -> ((string * Item.item) * Pcfg.t_bank) list -> string -> unit
```
Summary of surprisal values for quiet mode.

```
val pp_surp_sum_kilbury :
  'a -> ((string * Item.item) * Pcfg.t_bank) list -> string -> unit
```
Summary of surprisal values for experimental kilbury parsing in quiet mode.

```
val pp_short_sum :
  'a -> ((string * Item.item) * Pcfg.t_bank) list -> string -> unit
```
Total output for quiet mode.

```
val pp_short_sum_kilbury :
  'a -> ((string * Item.item) * Pcfg.t_bank) list -> string -> unit
```
Total output for quiet mode in experimental kilbury parsing.

4.4.3 Module `Print`

Low-level printing capabilities for standard out and file output.

```
val string_of_span : Item.span -> string
```
Return a string of a given span.

```
val string_of_spanlist : Item.span list -> string
```
Return a string of a given span list.

```
val string_of_anonitem : Item.span list list -> string
```
Return a string of a given span list list. Not used.

```
val string_of_item : Item.item -> string
```
Return a string of a given item.

```
val string_of_point : Point.point -> string
```
Return a string of a given pointer.

```
val print_item : (Item.item, Point.point) Hashtbl.t -> Item.item -> unit
```
Print a given item to standard out.

```
val string_label : string -> Item.span list -> string
```

```
val cat_label : 'a -> ('a, string) Hashtbl.t -> string
```

```
val full_label :
  string -> Item.span list -> 'a -> ('a, string) Hashtbl.t -> string
```

```
val qtree_of_tree : (string * string) Utilities.disptree -> string
```
For a given derivation/derived tree return a qtree-format string for LaTeX.

```
val dot_of_tree : (string * string) Utilities.disptree -> string
```
For a given derivation/derived tree return a qtree-format string for DOT.

```
val pp_of_tree : ('a * string) Utilities.disptree -> string
```
For a given derivation/derived tree return a pretty-printed string.

```
val print_trees :
  ('a ->
   string ->
   'b ->
   ((< pointers : ('d, 'e) Hashtbl.t; .. > as 'c) ->
    string -> 'f -> string list -> 'g list) ->
   'f -> 'h option -> 'c) ->
  'a ->
  string ->
  'b ->
  ('c -> string -> 'f -> string list -> 'g list) ->
  ('g -> string) -> 'f -> string -> unit
```
Given a checking function and a print function, return the appropriate display tree.

```
val print_trees_iter :
  ('a ->
   string ->
   (< pointers : ('c, 'd) Hashtbl.t; .. > as 'b) ->
```

```
   ('e -> string -> 'f -> string list -> 'g list) -> 'f -> 'h option -> 'e) ->
  'a ->
  string ->
  'b ->
  ('e -> string -> 'f -> string list -> 'g list) ->
  ('g -> string) -> 'f -> string -> unit
```

Same as above, but iterates over a list of sentences passed in from a file

```
val print_debug : ('a -> 'b -> 'c -> 'd) -> 'a -> 'b -> 'c -> 'e -> 'd
```

Given a checking function and a print function, return debugging statements from the stack during parsing.

References

Daniel M. Albro. An earley-style recognition algorithm for mcfgs.

K. Vijay-Shanker Avarind K. Joshi and David Weir. The convergence of mildly context-sensitive grammars. In S. M. Shieber and T. Wasow, editors, *The Processing of Natural Language Structure*. MIT Press, 1992.

Sylvie Billot and Bernard Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th annual meeting on Association for Computational Linguistics*, pages 143–151. Association for Computational Linguistics Morristown, NJ, USA, 1989.

Zhiyi Chi. Statistical properties of probabilistic context-free grammars. *Computational Linguistics*, 25(1):131–160, 1999.

Josh Goodman. Semiring parsing. *Computational Linguistics*, 25(4):573–605, 1999.

Ulf Grenander. *Syntax-controlled Probabilities.* Division of Applied Mathematics, Brown University, 1967.

Matthieu Guillaumin. Conversions between mildly context sensitive grammars. 2004.

M. Fujii H. Seki, T. Matsumura and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 1991.

John T. Hale. A probabilistic Earley parser as a psycholinguistic model. In *Proceedings of NAACL*, volume 2, pages 159–166, 2001.

John T. Hale. Uncertainity About The Rest of The Sentence. *Cognitive Science*, 2006.

Frederick Jelinek and John D. Lafferty. Computation of the probability of initial substring generation by stochastic context-free grammars. *Computational Linguistics*, 17(3):315–323, 1991. ISSN 0891-2017.

Avarind K. Joshi. Tree Adjoining Grammars: How Much Context-Sensitivity Is Required to Provide Reasonable Structural Descriptions? *Natural Language Parsing: Psychological, Computational and Theoretical Perspectives*, 1985.

Laura Kallmeyer. *Parsing Beyond Context-Free Grammars*. Not Avail, 2010. ISBN 364214845X.

Bernard Lang. Parsing incomplete sentences. In *Proceedings of the 12th conference on Computational linguistics-Volume 1*, pages 365–371. Association for Computational Linguistics Morristown, NJ, USA, 1988.

C.D. Manning, H. Schütze, and MIT Press. *Foundations of statistical natural language processing*. MIT Press, 1999.

R. Nakanishi, K. Takada, and H. Seki. An efficient recognition algorithm for multiple context free languages. *Proceedings of the Fifth Meeting on Mathematics of Language, MOL5*, 1997.

Mark-Jan Nederhof and Giorgio Satta. Estimation of consistent probabilistic context-free grammars. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 343–350. Association for Computational Linguistics Morristown, NJ, USA, 2006.

Mark-Jan Nederhof and Giorgio Satta. Computing partition functions of PCFGs. *Research on Language & Computation*, 6(2):139–162, 2008. ISSN 1570-7075.

Timothy J. O'Donnell, Noah D. Goodman, and Josh B. Tenenbaum. Fragment Grammars: Exploring Computation and Reuse in Language. 2009.

Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge Univ Pr, 1996. ISBN 052156543X.

Stuart M. Shieber, Yves Schabes, and Fernando C.N. Pereira. Principles and implementation of deductive parsing. *The Journal of Logic Programming*, 24(1-2):3–36, 1995.

Edward P. Stabler. Derivational minimalism. In *Logical Aspects of Computational Linguistics.*, pages 68–95. Springer, 1997.

Mark Steedman. *The Syntatcic Process*. MIT Press, 2000.

Andreas Stolcke. An efficient probabilistic context-free parsing algorithm that computes prefix probabilities. *Computational Linguistics*, 21(2):165–201, 1995.