massachusetts institute of technology — artificial intelligence laboratory

An Algorithm for Bootstrapping Communications

Jacob Beal

AI Memo 2001-016 August 13, 2001

© 2001 massachusetts institute of technology, cambridge, ma 02139 usa — www.ai.mit.edu

Abstract

I present an algorithm which allows two agents to generate a simple language based only on observations of a shared environment. Vocabulary and roles for the language are learned in linear time. Communication is robust and degrades gradually as complexity increases. Dissimilar modes of experience will lead to a shared kernel vocabulary.

1 Introduction

Neuroscience has postulated that the brain has many "organs" — internal subdivisions which specialize in one area. If we accept this view, then we need some sort of mechanism to interface these components. The design of this mechanism is limited by the hardware which the brain is constructed out of, as well as the size of the blueprints specifying how it is built. Neurons, as hardware, are relatively slow and imprecise devices, but they are very cheap, and it's easy to throw a lot of them at a problem in parallel. Our DNA is only about 1 gigabyte, too small to encode the full complexity of interfaces between all of the different components.

I approached this design problem from a hardware hacking point of view, with the question, "If I were designing the human brain, how would I build this interface?" It needs to be self-configuring, to beat the limited blueprints problem, and it needs to learn quickly. On the other hand, hardware is very cheap, and I can design in a domain with a huge number of interface wires between two components.

I have developed an algorithm which bootstraps communications solely from shared experience and I present it here as an existence proof and a tool for thinking about how a brain might be composed out of independent parts that learn to communicate with each other: it is possible for two agents to rapidly construct a language which enables them to communicate robustly.

2 System Model

There are two agents in the system, connected by a very large bundle of wires called *comm lines*. Each agent has another set of wires called *feature lines*, over which external information can arrive. Metaphorically, the comm lines are a nerve bundle connecting two regions of the brain and the feature lines

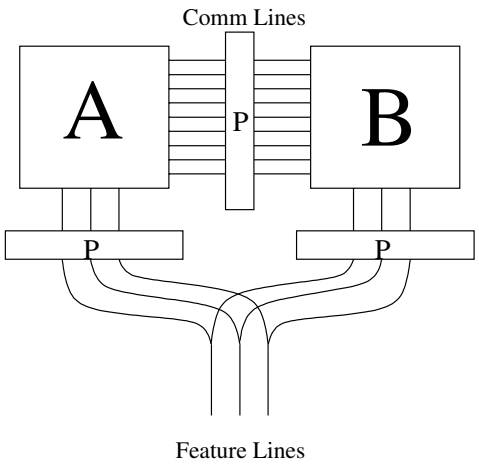


Figure 1: The agents labelled A and B are interconnected by *comm lines* — a bundle of wires with an arbitrary and unknown permutation. The agents also share some *feature lines* with the outside world, again with unknown permutations.

are nerve bundles that carry part of the brain's observation of the outside world. The actual wires in the bundles might be arbitrarily twisted and rearranged between the two ends of the system, so we add an unknown permutation to each bundle to model this effect and prevent any implicit sharing of ordering information between the two agents.

The comm lines have four states: 1,-1,0, and X. When undriven, the line reads as 0. Information is sent over the line by driving it to 1 or -1, and if the line is being driven to both 1 and -1, it reads as a conflict — X. In the experiments conducted here, I used a set of 10,000 comm lines.

Feature lines are named with a symbol which they represent and read as undriven, driven, or driven with another symbol. In the experiments I conducted, names of feature lines are things or actions and the symbols driven on them are roles. So typical feature lines might be **bob**, **mary**, or **push**, and typical roles might be **subject**, **object**, or **verb**. The set of feature lines is of undefined size — the agents have no knowledge of what feature names or roles exist until they encounter them in practice.

An agent can read and drive both comm and feature lines, but are constrained to synchronous schedule. Each cycle of the system has a "talk" phase and a "listen" phase. Agents can read lines at any time, but can only

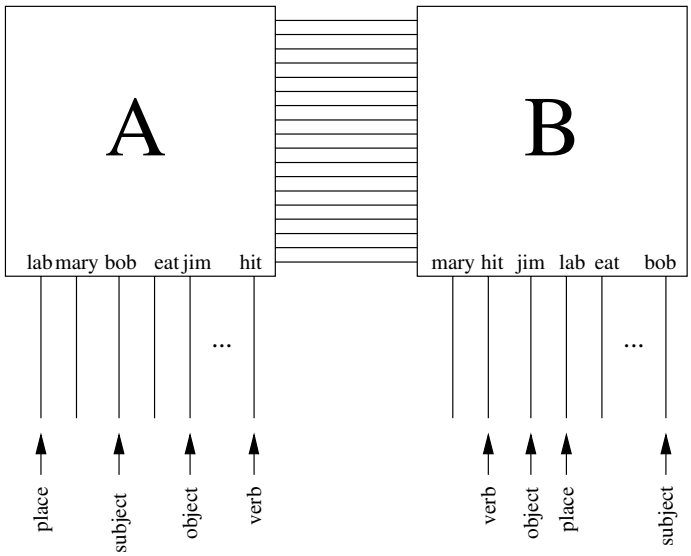


Figure 2: During a training cycle, the feature lines of both units are driven. Each agent attempts to learn from the comm lines driven by the other agent.

drive comm lines during the talk phase and feature lines during the listen phase. At the end of the talk phase, for symmetry breaking purposes one agent is randomly selected to have spoken first. The agent which spoke first can read the results of both agents speaking in its listen phase, while the one which spoke second reads only what the first agent spoke.

There are two types of cycles — training cycles and test cycles. In a training cycle, the data on the feature lines is sent to both agents. In a test cycle, one agent is randomly selected to receive input from the feature lines, while the other receives no input. Performance may then be evaluated on the basis of how well the output of the agent receiving no input matches the values on the feature lines.

2.1 Encoding View

Communication in this system may be viewed more abstractly as an encoding problem. The set of possible messages M are all sets of ordered pairs (A_1, A_2), where A_1 has a large range and A_2 has a small range. The challenge is to find an encoding system such that any M can be transmitted in a single

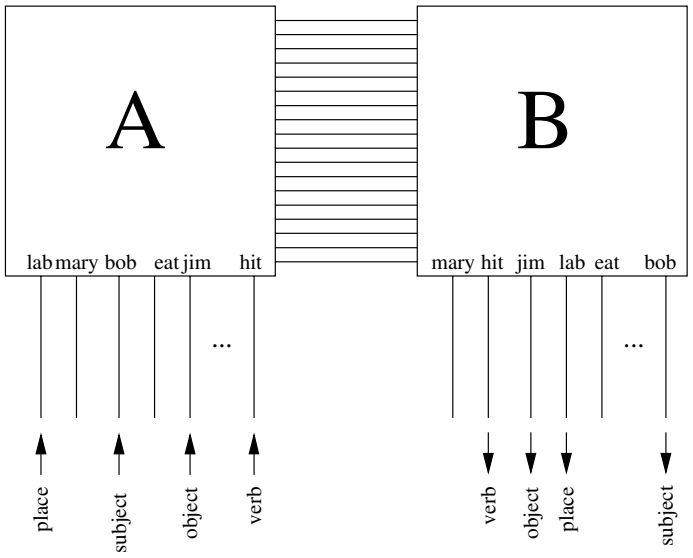


Figure 3: During a test cycle, the feature lines of one unit, say A, are driven and the feature lines of the other unit are observed. The test is scored by number of mistakes in B's reproduction of A's feature lines.

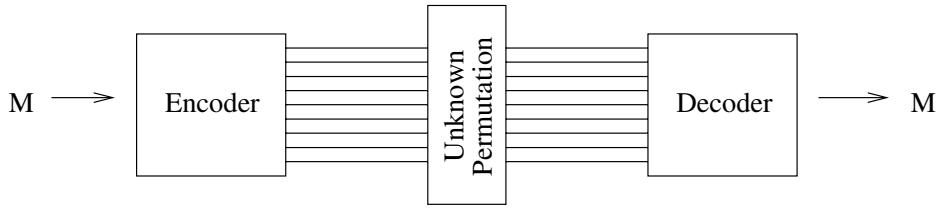


Figure 4: The communication system viewed as an encoding problem: the goal is to train the encoder and decoder to correctly communicate M from input to output despite the unknown permutation between them.

clock cycle and reconstructed by the decoder. In the specific instance I am addressing, then, A_1 is the vocabulary and A_2 is the role.

3 Algorithm

The key idea driving this algorithm is that sparseness makes it easy to separate the stimuli.

Knowledge in the system is represented by two sets of mappings: symbol mappings and inflection mappings. An inflection mapping links a symbol carried on a feature line to a real value between 0 and 1. A symbol mapping links a feature line with two sets of comm lines, designated as *certain* and *uncertain*, and includes an integer designated *certainty*.

These mappings are used symmetrically for production and interpretation of messages. In the "talk" phase, each driven feature line selects the *certain* comm lines associated via the symbol mapping and drives them with the unary fraction associated with the symbol on the feature line via the inflection mapping. In the "listen" phase, if enough of a feature line's associated comm lines are driven, then the feature line is driven with any inflection mapping symbols within a fixed radius of the unary code on that set of comm lines.

Both types of mappings are generated randomly when a feature or inflection is first encountered, then adjusted based on observations of the other agent's transmissions. These adjustments take place only if an agent spoke second; if it was the first one to speak, then its own transmissions are on the lines as well, inextricably mixed with the transmissions of the second agent, and this would make accurate learning significantly more difficult.

Inflection mappings are adjusted with a very simple agreement algorithm: if the received unary code is significantly different from expected, the code in the mapping is set to the received value. If a unary code matches which should not have, then it is purged and generated anew.

Symbol mappings are slightly more complicated. The first time an agent hears a given symbol spoken by the other agent, it adds every driven comm line to the *uncertain* lines for that symbol. Each time thereafter that it hears the symbol again, it intersects the driven lines with its *uncertain* lines, thereby eliminating lines associated with other symbols. After several iterations of this, it assumes that there is nothing left but lines which should be associated with the symbol, adds the *uncertain* lines to the *certain* lines, and begins to use them for communication. A few more iterations after that, it

begins paring down the *certain* lines the same way, so that the two agents can be assured that they have identical mappings for the symbol.

Formal Automaton Description:
Constants:
(Note that the precise values of these constants was arbitrarily chosen, and the behavior of the algorithm should be insensitive to small changes. For example, $p_s = 0.8$ just means ''a pretty good match'' and $t_c = 4$ means ''a few times'')
r_i = 0.05 (Radius of an inflection value)
p_s = 0.8 (Percent stimulus required to match a symbol)
t_c = 4 (Threshold where *uncertain* lines become *certain*)
t_p = 6 (Threshold to prune *certain* lines)
n_w = 10000 (Number of comm lines)
n_{wps} = 100 (Number of comm lines randomly selected for a new symbol)
w_m = 20 (Minimum number of comm lines per symbol)

Input:
$talk_{in}(F)$
F is a set of (s, i), where s, i are symbols (feature lines)
$listen_{in}(C_i, F, first)$
C_i, $0 < i \leq n_w$, $\in \{1, 0, -1, X\}$ (comm lines)
$first$ is boolean, F as for $talk_{in}$

Output:
$talk_{out}(C_i)$, C_i as for $listen_{in}$
$listen_{out}(F)$, F as for $talk_{in}$

States:
T_s is a set of $x = (x_s, x_c, x_u, x_n)$ where x_s is a symbol,
x_c, x_u are sets of $r \in N^+$, $0 < r \leq n_w$, and $x_n \in N^+$
initially $\emptyset$
T_i is a set of $y = (y_i, y_v)$ where y_i is a symbol and $0 \leq y_v \leq 1$
initially $\emptyset$
c is a set of (l, v) where $l \in N^+$, $0 < l \leq n_w$ and $v \in \{1, 0, -1, X\}$, initially $\emptyset$,
with the join rules $(l, 1) \cup (l, -1) = (l, X)$, and $(l, X) \cup (l, *) = (l, X)$
f is a set of (s, i) where s, i are symbols, initially empty
$talking, listening$ are booleans, initially false

Transitions:

$talk_{in}(F)$

```
  Effect:
    talking := true
    ∀ (s,i) ∈ F
      if not ∃ x ∈ T_s s.t. x_s = s
        T_s := T_s ∪ (s,set of n_wps random elements,∅,0)
      if not ∃ y ∈ T_i s.t. y_i = i
        T_i := T_i ∪ (i,random)
      let x ∈ T_s s.t. x_s = s
           y ∈ T_i s.t. y_i = i
           v_l ∈ {1,-1}, ∀ l ∈ x_c s.t.
                precisely |x_c| * y_v of the set {v_l} are 1
        ∀ l ∈ x_c
          c := c ∪ (l, v_l)
```

$talk_{out}(C_i)$

```
  Preconditions:
    ∀ (l,v) ∈ c
      C_l = v
    ∀ l s.t. ∀ v, (l,v) ∉ c
      C_l = 0
    talking = true
  Effect:
    talking := false
    c := ∅
```

$listen_{in}(C_i, F, first)$

```
  Effect:
    listening := true
    ∀ m ∈ T_s
      if not first and ∃ (s,i) ∈ F s.t. s = m_s
        if m_n = 0
          m_n := 1
          m_u := {j | C_j ≠ 0}
        else
          m_n := m_n + 1
```

```
        if m_n < t_c
            m_u := m_u ∩ {j | C_j ≠ 0}
        if t_c ≤ m_n < t_p
            m_c := m_c ∪ m_u
            m_u := ∅
        if t_p ≤ m_n
            if |m_c ∩ {j | C_j ≠ 0}| ≥ w_m
              m_c := m_c ∩ {j | C_j ≠ 0}
            else
              T_s := T_s − m
    else
      let x = m_c ∩ {j | C_j ≠ 0}
            u = |{j ∈ x | C_i = 1}|/|{j ∈ x | C_j ∈ {1, −1}}|
        if |x| ≥ |m_c| * p_s
            if ∃ y ∈ T_i s.t. |y_v − u| < r_i
              ∀ y ∈ T_i s.t. |y_v − u| < r_i
                 f := f ∪ (m_s, y_i)
            else
              f := f ∪ (m_s, null)
  if not first
    ∀ y ∈ T_i
      if ∃ (s, i) ∈ F s.t. i = y_i and ∃ m ∈ T_s s.t. m_s = s
        let u = |{j ∈ m_c | C_j = 1}|/ |{j ∈ m_c | C_j ∈ {1, −1}}|
            if |u − y_v| > r_i/2 and {j ∈ m_c | C_j = X} = ∅
              y_v := u
      else
        if ∃ (s, i) ∈ F, m ∈ T_s s.t.
              |(|{j ∈ m_c | C_j = 1}|/ |{j ∈ m_c | C_j ∈ {1, −1}}|) − y_v| < r_i
              and {j ∈ m_c | C_j = X} = ∅
            y_v := random
    ∀ y ∈ T_i
      if ∃ z ∈ (T_i − y) s.t. |z_v − y_v| < r_i * 2
        T_i := T_i - (random ∈ {y, z})

listen_out(F)
  Preconditions:
    F = f
    listening = true
```

```
Effect:
   listening := false
   f := ∅
```

See Appendix A for Scheme code implementing the algorithm.

4 Results

To test the algorithm, I used a system with an n_w of 10000 comm-lines and a n_{wps} of 100 random wires selected to generate a new symbol mapping.

I trained the system for 1000 cycles, then evaluated its performance over an additional 200 cycles. Each cycle, an example is generated and presented to the system. In the training phase, there is an 80% chance it will be presented to both agents and a 20% chance it will be presented to only one (That is, 80% training cycles, 20% test cycles). During the evaluation phase, the first 100 are presented to the first agent only, and the second 100 are presented to the second agent only. A test is considered successful if the input feature set is exactly reproduced by the listening agent.

The examples input to the feature lines are thematic role frames generated from a set of 50 nouns, 20 verbs, and 4 noun-roles. Each example is randomly generated with 0-2 verbs assigned the "verb" role and 2-4 nouns assigned noun-roles. No noun, verb, or noun-role can appear more than once in an example. A typical scene, then, might be `'((approach verb) (jim subject) (shovel instrument) (lab object))`, which corresponds loosely to "Jim approached the lab with the shovel." All told, there are more than 1.2 billion examples which can be generated by the system, so in general an agent will never see a given scene twice.

In a typical run of this system, after about 200 cycles most symbols will have entered the shared vocabulary and can be successfully communicated between the two agents. After about 500 cycles, the set of inflections will have stabilized as well. In the final round of 200 tests, the success rate is usually 100%, although occasionally due to the stochastic nature of the algorithm, the inflections will not yet have converged by the end of 1000 tests and consequently one or more will not be transmitted correctly.

4.1 Convergence Time

The time needed to develop a shared vocabulary is proportional to the number of symbols in the vocabulary. A symbol is learned when both agents have *certainty* for that symbol greater than t_c. An agent increases *certainty* when it speaks second, which is determined randomly, so we may estimate this as a Poisson process. Thus, we may calculate the expected number of cycles, c, as follows:

$$E(c) = 2t_c \frac{\binom{2t_c}{t_c}}{2^{2t_c}} + \sum_{n=2t_c+1}^{\infty} n \frac{\binom{n-1}{t_c-1}}{2^{n-1}}$$

Evaluating this for $t_c = 4$, we find an expectation of 10.187 uses of a symbol before both *certainty* thresholds are reached.

For these experiments then, with an average of 3 nouns and 1 verb per training cycle, then, we can calculate the expected number of shared symbols S as a function of elapsed cycles t:

$$S(t) = n_{nouns} * (1 - P(10.187, t\frac{3}{n_{nouns}} 0.8)) + n_{verbs} * (1 - P(10.187, t\frac{1}{n_{verbs}} 0.8))$$

where P is the incomplete gamma function. Since this function is linear in the number of symbols, we see that the time to build a shared vocabulary is linear in the number of symbols. Figure 5 shows experimental data confirming this estimate.

Once a shared vocabulary of symbols exists, the algorithm can begin learning inflections. If n_i is the number of inflections to be learned, and r_i is chosen such that $r_i * n_i \leq 0.5$, then we can show that the time to develop a shared set of inflections is $O(n_i)$.

An inflection may be learned any time a symbol is successfully transmitted in a training cycle. This occurs if the new inflection does not conflict with any of the previously learned inflections - that is, if n symbols have already been learned, then it must be the case that for all v_i s.t. $1 \leq i \leq n$, $|v_{n+1} - v_i| < 2r_i$. Since the value of the new symbol, v_{n+1}, is chosen by a uniform random process on the interval $[0, 1]$, the probability p_{n+1} of choosing an acceptable inflection value is no less than $1 - (2r_i * n)$. The n_ith inflection, then, has the least probability of success, $p_{n_i} = 1 - (2r_i * (n_i - 1)) \geq 2r_i$, and p_n is generally bounded below by $2r_i$.

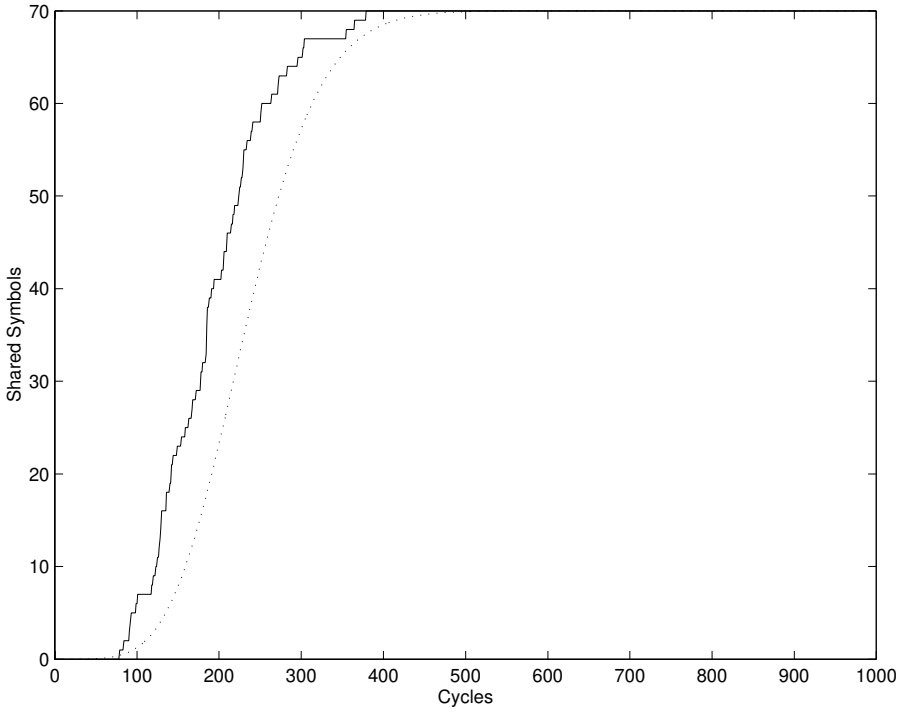


Figure 5: Number of shared symbols versus elapsed time for 50 nouns and 20 verbs. Dotted line is theoretical estimate S(t), solid line is experimental data.

For these experiments then, we can calculate the expected number of inflections, assuming a shared vocabulary, as a function $I(t)$ of elapsed cycles t. There are expected to be 3 noun inflections and 1 verb inflection per training cycle, so the least frequent inflection is expected to appear at with frequency at least $1/n_i$. Thus, we obtain

$$I(t) = n_i * (1 - P(1, 2r_i t\frac{1}{n_i}0.8))$$

where P is the incomplete gamma function. Since this function is linear in the number of inflections, we see that the time to build a shared set of inflections is linear in the number of inflections. Figure 6 shows experimental data confirming this estimate.

Thus, the algorithm is expected to converge in $O(s + n_i)$ time, where s is the size of the vocabulary and n_i is the number of inflections.

4.2 Channel Capacity

The number of symbols and roles which can be learned without false symbol detection and inflection misinterpretation is dependent on the number of wires n_w, the number of wires per symbol n_{wps}, and the percent stimulus necessary to recognize a symbol p_s.

If we want no combination of symbols to be able to generate a spurious recognition, then each symbol must have at least $n_{wps}(1 - p_s)$ wires not used by any other symbol. This means that a vocabulary would have a maximum size of only $\frac{n_w}{n_{wps}(1-p_s)}$. In practice, however, we can assume that only a few symbols are being transmitted simultaneously. If we assume that no more than m symbols will be transmitted at once, then we can conservatively estimate capacity by allowing any two symbols to overlap by no more than $n_{wps} * p_s/m$ wires. Thus any given symbol covers a portion of symbol space with volume:

$$\sum_{i=0}^{n_{wps}(1-\frac{p_s}{m})} \binom{n_{wps}}{i} \binom{n_w - n_{wps}}{i}$$

The whole symbol space has volume $\binom{n_w}{n_{wps}}$, so a conservative estimate of the maximum number of symbols that can exist is:

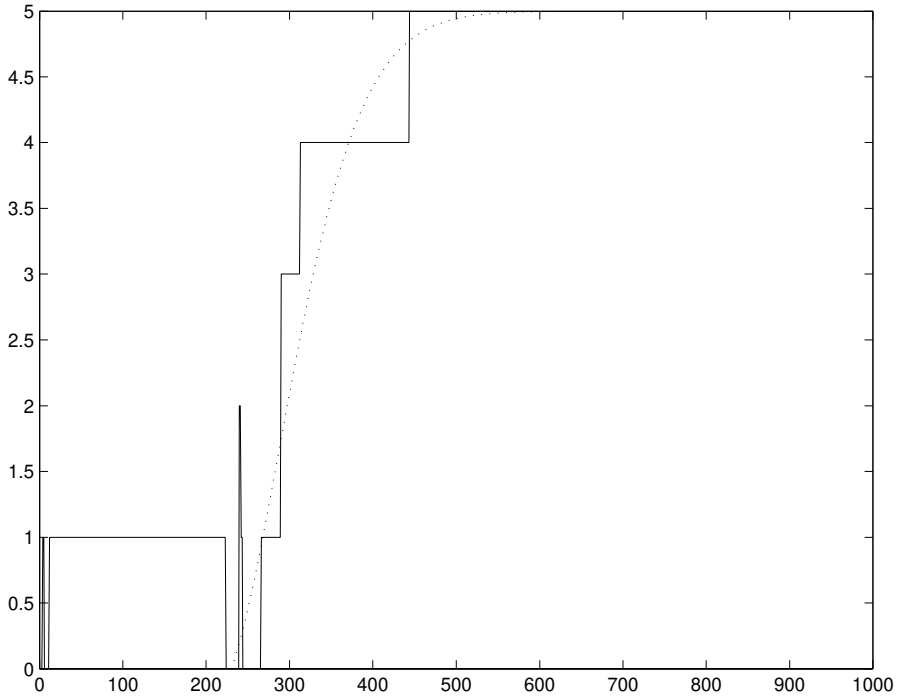


Figure 6: Number of shared inflections versus elapsed time for 4 noun inflections and 1 verb inflection, in a system with 50 nouns and 20 verbs. The dotted line is theoretical estimate I(t), beginning with cycle 230, where S(t) predicts half the vocabulary to be learned. The solid line is experimental data.

$$\frac{\binom{n_w}{n_{wps}}}{\sum_{i=0}^{n_{wps}(1-\frac{p_s}{m})} \binom{n_{wps}}{i}\binom{n_w-n_{wps}}{i}}$$

This yields a satisfactorily large capacity for symbols. For the experiments described above, with $n_w = 10000$, $n_{wps} = 100$, $p_s = 0.8$ and a maximum of 6 concurrent symbols, we find that the capacity is 1.167×10^{12} distinct symbols.

4.3 Performance Degradation

We expect that the performance of the algorithm will degrade gracefully as the channel capacity is reduced. As the average Hamming distance between symbols drops, the chance that a combination of other symbols will overlap to produce a spurious recognition or interfere with the inflection being transmitted rises. Since symbols receiving too much interference are discarded, the algorithm will tend to break up clusters of symbols and move toward an efficient filling of symbol space. Thus, reducing the ratio n_w/n_{wps} ought to cause the transmission errors to rise gradually and smoothly. In practice we find that this is in fact the case, as shown in Figure 7.

4.4 Parameter Variation

The values of the parameters used in the experiments above were not carefully chosen. Rather, I made a guess at a reasonable value for each parameter, expecting that the algorithm should not be very sensitive to the parameter values. (If it were, then I could hardly claim it was a robust algorithm!)

To test this, I ran a series of experiments in which I trained and tested the system with one of the parameters set to a different value. For each value for each parameter I ran 10 experiments: Figure 8 shows the performance of the algorithm as a function of parameter value for each of the six parameters p_s, r_i, t_c, t_p, w_m, and n_{wps}. (n_w is excluded because its variation is evaluated in the preceding section) As predicted, the performance of the algorithm is good over a wide range of values for each variable.

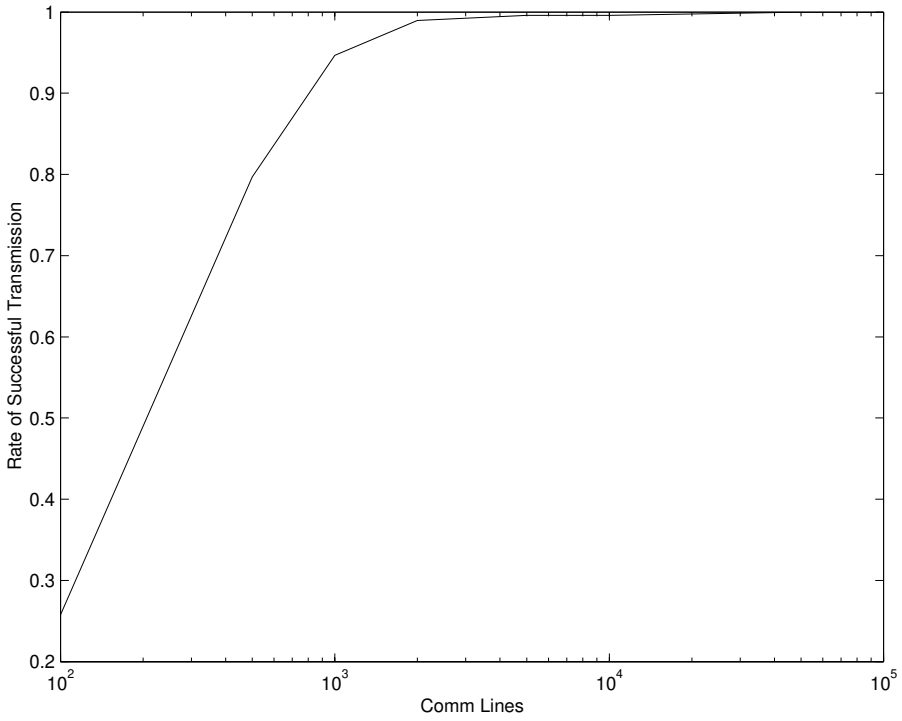
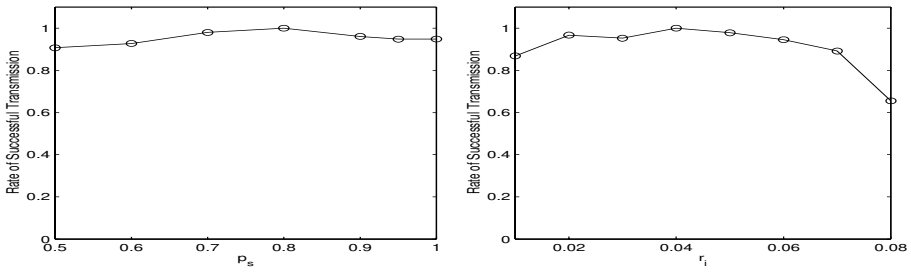
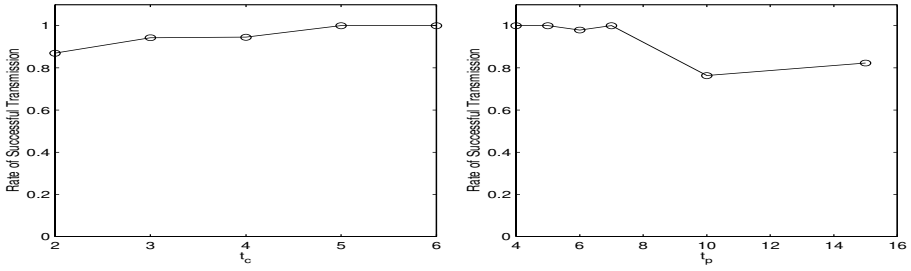


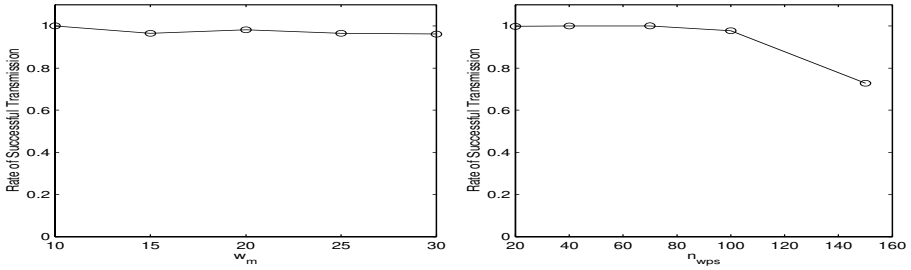
Figure 7: Number of comm lines versus transmission robustness. Horizontal axis is n_w from 100 to 100,000. Vertical axis shows the symbols and inflections correctly received per symbol and inflection transmitted (spurious receptions count against this as well) over the course of 200 test cycles on a system trained with 50 nouns, 20 verbs and 4 noun-roles, $n_{wps} = 20$, $p_s = 0.8$. Accuracy degrades smoothly with decreased channel capacity.



(a) Variation of p_s (b) Variation of r_i



(c) Variation of t_c (d) Variation of t_p



(e) Variation of w_m (f) Variation of n_{wps}

Figure 8: Variation in performance as each parameter is varied. For each graph, the horizontal axis shows the value of the parameter being varied and the vertical axis shows the fraction of symbols and inflections correctly received per symbol and inflection transmitted. Measurements are the average values over the course of 10 runs of 200 test cycles, as in 7. For each run of test cycles, the systems were trained with 50 nouns, 20 verbs and 4 noun-roles, with base parameter values $p_s = 0.8$, $r_i = 0.05$, $t_c = 4$, $t_p = 6$, $w_m = 20$, $n_{wps} = 100$, and $n_w = 10000$. All parameters in the system can tolerate small variations without serious degradation in performance.

4.5 Dissimilar Features

The operation of this algorithm is not confined to agents presented with identical features. If dissimilar but correlated feature sets are presented, then the similarities will be discovered and communicated in the language shared between the two agents. Features which are not shared will be transmitted as well, but will never be successfully interpreted by the listening agent.

I ran an experiment to confirm this, in which examples generated with 7 nouns, 6 verbs and 4 noun-roles were run through a filter before being input to the feature lines of each agent. The filter on the first agent added 0-2 random words from a set of 4, and split one noun into a pair of features. The filter on the second agent added precisely one random word from a set of four, remapped the verbs onto a set of overlapping component features, and relabelled one of the noun-roles. The system was then trained for 200 cycles with $n_w = 1000$ and $n_{wps} = 20$.

The final results showed that, as expected, the system had learned a shared vocabulary despite the handicaps imposed. The random words added to each side were ignored, since their appearance in the features of one agent was uncorrelated to the features seen by the other agent. The noun consistently split into a pair of features for the first agent was interpreted as a single symbol by the second agent, and communicated successfully in both directions. The relabelled noun-role in the second agent was also communicated unerringly. The verbs remapped onto component features for the second agent, however, were not able to be fully communicated. This was not unexpected, and occurred as predicted. The verb remapping was as follows:

$$move \rightarrow go$$

$$retreat \rightarrow go, from$$

$$approach \rightarrow go, to$$

$$touch \rightarrow go, contact$$

$$eat \rightarrow ingest, contact$$

$$fear \rightarrow fear$$

Since symbols are differentiated by intersection, the translation of several different symbols to the "go" symbol prevented "go" from being successfully communicated in either direction. Similarly, "contact" was impaired by its

mapping to both "eat" and "touch". As a result, "move" and "touch" could not be reliably communicated. All of the other verbs, however, having some unique component, were able to be communicated between the two agents successfully.

5 Contributions

I have built an algorithm which allows two agents to generate a shared language on the basis of shared experiences only. I fully expect that this algorithm can be greatly improved upon.

6 Acknowledgements

Particular note should be given to the help from several people. Gerry Sussman started me thinking about the problem and pointed me in this direction. My research is part of a continuing effort started by Yip and Sussman to build a "TTL databook for the mind" — a compatible system of modules that capture aspects of mental activity and can be combined to build ever more powerful systems. Thanks also to Catherine Havasi for hacking some of the early code with me and being a needling presence to keep me from slacking off.

References

[1] Kirby, Simon. "Language evolution without natural selection: From vocabulary to syntax in a population of learners." Edinburgh Occasional Paper in Linguistics EOPL-98-1, 1998. University of Edinburgh Department of Linguistics.

[2] Kirby, Simon. "Learning, Bottlenecks and the Evolution of Recursive Syntax." Linguistic Evolution through Language Acquisition: Formal and Computational Models edited by Ted Briscoe. Cambridge University Press, in Press.

[3] Lynch, Nancy. Asynchronous Systems Model. In *Distributed Algorithms*, Morgan Kaufmann Publishers, Inc, San Francisco, California, 1996. Chapter 8, pages 199-234.

[4] Minsky, Marvin. *The Society of Mind*. Simon & Schuster, Inc, New York, 1985.

[5] Mooers, Calvin. "Putting Probability to Work in Coding Punched Cards: Zatocoding (Zator Technical Bulletin No. 10), 1947. Reprinted as Zator Technical Bulletin No. 65 (1951).

[6] Yip, Kenneth and Sussman, Gerald Jay. "Sparse Representations for Fast, One-Shot Learning." MIT AI Lab Memo 1633, May 1998.

A Scheme Code

A.1 Testbed

```
;; runs simple tests through:

(declare (usual-integrations))

(cf "utils.scm")
(cf "talk2.scm")
(cf "talker.scm")
(cf "runtest.scm")

(load "utils")
(load "talk2")
(load "talker")
(load "runtest")

(define (make-log name)
  (let ((fout (open-output-file name)))
    (lambda (x) (if (eq? x 'flush-buffer)
                    (flush-output fout)
                    (pp x fout)))))
;; exa of use:
;; (define thelog (make-log "thefile.out"))
;; (thelog '(this is a list I want to log))
;; To flush to disk:
;; (thelog 'flush-buffer)
```

```
;; The system consists of two talkers, each with a feature-bus coming
;; in, and a comm-line connecting.
;; The state of this system can be described as a list of five elements:
;; (feat1 talker1 comm talker2 feat2)
;; Starting pattern for a instance
;; ( ((f1 1) (f2 2)) imap1 () imap2 () )
;;        or
;; ( ((f1 1) (f2 2)) imap1 () imap2 ((f1 1) (f2 2)) )
;; Run a test by:
;;   update imap1, update imap2: check for quiescence


;; A feature set is defined as follows:
;; There are five roles which can be expressed:
;; subject, object, action, instrument, place
;; Actions come from category "verb" and anywhere from 0-2 can be asserted
;; All others are nouns and 2-4 can be asserted
;; (YES, IT'S PRETTY ARBITRARY: I JUST WANT A DECENT-SIZED SPACE)
;; How big is it? 28 nouns, 6 verbs. avg 4 words/sentence.

; Normal Set
;(define nouns '(bob jim mary icepick shovel table lab))
;(define verbs '(move approach retreat touch eat fear))
;(define noun-roles '(subject object instrument place))

; Mega Set - 50 nouns, 20 verbs
(define nouns '(bob jim mary icepick shovel table lab fred bill classroom
                leg cup butterfly dog cat turtle door window car hammer
                keyboard coffee danish pencil pen eraser wall socket book ed
                wheel chainsaw gun kite bedroom shower beach shoe light dark
                hat office house apple banana flea vampire stapler kim joe))
(define verbs '(move approach retreat touch eat fear zap feel fly throw
                catch push hit stab tickle hurt love hate want ignite))
(define noun-roles '(subject object instrument place))
```

```
(define (generate-features)
  (let loop ((nnoun (+ 2 (random 3)))
             (nverb (random 3))
             (features '())
             (roles '()))
    (let ((vp (list-transform-negative verbs (lambda (x) (memq x features))))
          (np (list-transform-negative nouns (lambda (x) (memq x features))))
          (nr (list-transform-negative noun-roles (lambda (x) (memq x roles)))))
      (if (<= nnoun 0)
          (if (<= nverb 0)
              (map (lambda (x y) (if y
                                     (list x y 1)
                                     (list x 'verb 1)))
                   features roles)
              (loop 0 (- nverb 1)
                    (cons (list-ref vp (random (length vp))) features)
                    (cons #f roles)))
          (loop (- nnoun 1) nverb
                (cons (list-ref np (random (length np))) features)
                (cons (list-ref nr (random (length nr))) roles))))))


(define cycle-log (make-log "cycle.log"))

(define (run-cycles initial-state i terminate)
  (let* ((features (generate-features))
         (istest (< 0.8 (random 1.0)))
         (testwhich (< 0.5 (random 1.0)))
         (state (list (if istest (if testwhich features #f) features)
                      (second initial-state)
                      #f
                      (fourth initial-state)
                      (if istest (if testwhich #f features) features))))
    (let ((newstate (iterate state)))
      (if istest
          (if testwhich
              (pp (list i 'test2 (first state) 'returns (fifth newstate)))
              (pp (list i 'test1 (fifth state) 'returns (first newstate))))
```

```
          (pp (list i 'train (first state))))
      (cycle-log (list 'step i features state newstate))
      (cycle-log 'flush-buffer)
      (if (or (eq? terminate -1) (>= i terminate))
          newstate
          (run-cycles newstate (+ i 1) terminate)))))

(define cyclestate (run-cycles '(#f #f #f #f #f) 1 1000)) ;; 300

;; test a percent-correct measure
(let loop ((i 0) (successes 0) (state cyclestate))
  (let* ((features (generate-features))
         (state (list (if (< i 100) features #f)
                      (second state)
                      #f
                      (fourth state)
                      (if (< i 100) #f features))))
    (if (>= i 200)
        (pp `(Final results: ,successes sucesses out of ,i trials))
        (let ((newstate (iterate state)))
          (let ((fout (if (< i 100) (fifth newstate) (first newstate))))
            (if (and (equal-set? features (fifth newstate) equal?)
                     (equal-set? features (first newstate) equal?))
                (begin
                  (cycle-log (list 'test i features 'succeed))
                  (cycle-log 'flush-buffer)
                  (loop (+ i 1) (+ successes 1) newstate))
                (begin
                  (cycle-log (list 'test i features fout 'fail))
                  (cycle-log 'flush-buffer)
                  (loop (+ i 1) successes newstate)))))))))

(declare (usual-integrations))

(define (make-log name)
  (let ((fout (open-output-file name)))
    (lambda (x) (if (eq? x 'flush-buffer)
                    (flush-output fout)
```

```
                    (pp x fout)))))


(define get-feat1 first)
(define get-imap1 second)
(define get-comm third)
(define get-imap2 fourth)
(define get-feat2 fifth)

;; talk returns: (imap comm)
;; prune-conflicts returns: imap
;; listen returns: (imap feat)
;; iterate operates in the following manner:
;; 1. comm lines start blank.
;; 2. have everything speak
;; 3. have everything resolve speaking conflicts
;; 4. have everything listen

;; Unfortunately, if we do this in perfect order, then the synchrony
;; makes it difficult to separate the outgoing and incoming transmissions,
;; most particularly, to determine which bits of a feature are correct.
;; We resolve this by ordering the transmissions such that one of the
;; two hears the others transmissions before its transmission clutters
;; the wires. (If, of course, only one is transmitting, then it cannot
;; be pre-empted)

(define (iterate state)
  (let* ((tres1 (talk '() (get-feat1 state) (get-imap1 state)))
         (tres2 (talk '() (get-feat2 state) (get-imap2 state)))
         (comms (combine-comms (second tres1) (second tres2)))
         (whofirst (if (and (second tres1) (second tres2))
                       (if (eq? (random 2) 0) #t #f)
                       (if (second tres1) #t #f)))
         (lres1 (listen whofirst (if whofirst comms (second tres2))
                        (get-feat1 state) (first tres1)
                        (get-imap1 state) (second tres1)))
         (lres2 (listen (not whofirst) (if whofirst (second tres1) comms)
                        (get-feat2 state) (first tres2)
```

```
                        (get-imap2 state) (second tres2)))
         (newstate (list (second lres1) (first lres1) comms (first lres2)
                         (second lres2))))
    newstate)) ; for now, just doing a single learning pass

;; this merges a set of comm lines, turning transmissions into noise
(define (combine-comms . inputs)
  (fold-left
   (lambda (x y)
     (let* ((res1 (split x
                         (lambda (a)
                           (list-search-positive y
                             (lambda (b) (equal? (first b) (first a)))))))
            (res2 (split y
                         (lambda (a)
                           (list-search-positive x
                             (lambda (b) (equal? (first b) (first a)))))))
            (merged (map (lambda (a)
                           (let* ((v2 (second
                                       (list-search-positive (first res2)
                                         (lambda (b)
                                           (equal? (first b) (first a)))))))
                             (list (first a) (if (eq? v2 (second a))
                                                 (second a)
                                                 'x))))
                         (first res1))))
       (append (second res1) (second res2) merged)))
   '()
   inputs))
```

A.2 Algorithm

```
;; Talker
;; Wires: only driven values are listed:
;; ( (line 1/-1/X) )
;; NOTE: Right now, all asserted feature lines are assumed to be positive

(declare (usual-integrations))
```

```
(define num-wires 10000) ;; 1000
(define num-wires-per-symbol 100) ;;20
(define min-wires-per-symbol 20) ;;5
(define percent-match 0.8)
(define unary-percent-match 0.05)

;; 12/6 Reformatting internal-map bundles as follows:
;; Instead of (symbol (commlines)) they now become
;; (symbol (certaincommlines) (uncertaincommlines) uncertainfactor)
;; When listening, both certain and uncertain comm lines are considered
;; When talking, only certain comm lines are used
;; An uncertain comm lines become certain after surviving n transmissions
(define certainty-threshold 4)
(define certain-pruning-threshold 6)

;; line abstractions
;; returns a line-bundle with the new value
;; On a conflict between values, the line becomes unasserted
;; This is represented by an "X" on the line
(define (assert-line line-bundle line value)
  (let* ((result (split line-bundle
                        (lambda (x) (eq? (first x) line))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if pos
        (if (equal? (second (car pos)) value)
            line-bundle
            (cons (list line 'x) neg))
        (cons (list line value) line-bundle))))

;; this can put inflections on lines as well as simple values
(define (assert-feature-line line-bundle line inflection value)
  (let* ((result (split line-bundle
                        (lambda (x) (eq? (first x) line))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if (and pos (or (not inflection)
```

```
                    (equal? (second (car pos)) inflection)))
        (if (or (not inflection) (equal? (third (car pos)) value))
            line-bundle
            (cons (list line inflection 'x) neg))
        (cons (list line inflection value) line-bundle))))

;; clean-bits: returns only the non-conflicted bits
(define (clean-bits line-bundle)
  (list-transform-negative line-bundle (lambda (x) (eq? (second x) 'x))))

;; returns the value found on the line, or zero if it's not asserted
;; Conlicted bits *are* returned as an "X"
;; Thus, there are four possible results: 1,-1,0,x
(define (test-line line-bundle line)
  (let ((result (list-search-positive line-bundle
                  (lambda (x) (eq? (first x) line)))))
    (if result (second result) 0)))

;; can check the inflection on feature-lines
(define (test-inflection line-bundle line)
  (let ((result (list-search-positive line-bundle
                  (lambda (x) (eq? (second x) line)))))
    (if result (second result) 0)))

;; listen & talk return (cons internal-map foo-line)
;; Two cases to handle:
;; 1. Features in internal-map
;;    If asserted on feature-lines: modify internal-map for conflicts
;;    else: if enough matches, assert on feature-lines
;; 2. Features asserted, but not in internal-map
;;    add new feature to internal-map
;; Returns (cons internal-map feature-lines
;; 1/2: added spokefirst
;; 1/4: changed "new features" to be features *heard* for the first time,
;;      regardless of whether they've been spoken before.
;;   NOTE: as it's set up, there's some overkill. Really, the critical
;;     factor is whether c-a is an empty set of not at symbol-creation.
;;     However, I'm keeping it this way since it's a bit more "pure"
```

```
(define (listen spokefirst comm-lines feature-lines internal-map pre-map comm-spok
  (let* ((smap (if internal-map (second internal-map) #f))
         (nmap (if internal-map (first internal-map) #f))
         (res (split smap
                (lambda (x)
                  (and (equal? 0 (test-line feature-lines (first x)))
                       (equal? 0 (test-inflection feature-lines (first x)))))))
         (stimfeat (second res))
         (smap-check (first res))
         (res2 (split stimfeat
                 (lambda (x)
                   (or spokefirst
                       (< 0 (find-symbol-certainty internal-map (first x)))))))
         (newfeat (second res2))
         (c-a (list-transform-negative comm-lines
                (lambda (x) (member x comm-spoke))))
         (newfeat
          (map (lambda (x)
                 (list (car x) (find-symbol-codes internal-map (car x))
                       (map first c-a) 1))
               (second res2)))
         (smap-assert (if (not spokefirst)
                          (listen-resolve-conflicts spokefirst comm-lines
                                                    (first res2))
                          (first res2))))
    (list (list (if (not spokefirst)
                    (nsquelch (update-nmap nmap comm-lines
                                           feature-lines stimfeat))
                    nmap)
                (append smap-check smap-assert newfeat))
          (listen-assert-features comm-lines feature-lines nmap smap-check))))




;; returns a new internal-map with the codes changed/added
;; If a code-mapping drops below a minimum size, it is considered
;; worthless and discarded. This can happen to either new codes or
```

```
;; codes being changed.
;; 12/6: modified to handle certainty
(define (set-symbol-codes imap symbol codes)
  (let* ((smap (if imap (second imap) #f))
         (nmap (if imap (first imap) #f)))
    (list nmap (set-smap-codes smap symbol codes))))

(define (find-smap-codes smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (second elt) #f)))

(define (find-smap-uncertains smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (third elt) #f)))

(define (find-smap-certainty smap symbol)
  (let* ((elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (fourth elt) #f)))

(define (set-smap-codes smap symbol codes)
  (let* ((result (split smap
                        (lambda (x) (eq? (first x) symbol))))
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (if (< (+ (length (first codes)) (length (second codes)))
           min-wires-per-symbol)
        (begin
          neg)
        (if pos
            (cons (cons symbol codes) neg)
            (cons (cons symbol codes) smap)))))

;; sets the code for an inflection
(define (set-inflection-code imap inflection ucode)
  (let* ((smap (if imap (second imap) #f))
         (nmap (if imap (first imap) #f))
         (result (split nmap
                        (lambda (x) (eq? (first x) inflection))))
```

```
         (pos (first result)) ;; should have 0 or 1 entries
         (neg (second result)))
    (list (cons (list inflection ucode) neg) smap)))

;; The purpose of this function is to shrink the symbols in the
;; association DB in order to have more precise interpretations
;; of which lines represent a given symbol. (NOTE: explain more clearly)
;; Implemented by:
;; 12/6: adds 1 to the certainty of each symbol.
;; 1/2: added spokefirst
;; Certain lines are not filtered until the symbol has become certain
(define (listen-resolve-conflicts spokefirst comm-lines mappings)
  (let loop ((map-list mappings) (new-imap mappings))
    (if map-list
        (let* ((elt (first map-list))
               (ccodes (find-smap-codes new-imap (first elt)))
               (cnewcodes (list-transform-negative ccodes
                            (lambda (x)
                              (eq? 0 (test-line comm-lines x)))))
               (ucodes (find-smap-uncertains new-imap (first elt)))
               (unewcodes (list-transform-negative ucodes
                            (lambda (x)
                              (eq? 0 (test-line comm-lines x)))))
               (newcert (+ (if spokefirst 0 1)
                           (find-smap-certainty new-imap (first elt))))
               (newcodes
                (if (>= newcert certainty-threshold)
                    (if (>= newcert certain-pruning-threshold)
                        (list (->uniset (append cnewcodes unewcodes))
                              '() newcert)
                        (list (->uniset (append ccodes unewcodes))
                              '() newcert))
                    (list ccodes unewcodes newcert))))
          (loop (cdr map-list)
                (set-smap-codes new-imap (first elt) newcodes)))
        new-imap)))

;; This functions asserts a feature line if enough of its associated
```

```
;; comm-lines are lit up
;; 12/6: Only certain comm lines are considered.
(define (listen-assert-features comm-lines feature-lines nmap mappings)
  (fold-left (lambda (features imap-elt)
               (let* ((ll (second imap-elt)) ;; line-list
                      (fll (list-transform-negative ll
                             (lambda (x)
                               (eq? 0 (test-line comm-lines x)))))
                      (num-1s (length (list-transform-positive ll
                                        (lambda (x)
                                          (eq? -1 (test-line comm-lines x))))))
                      (num1s (length (list-transform-positive ll
                                       (lambda (x)
                                         (eq? 1 (test-line comm-lines x)))))))
                 (if (and ll (>= (/ (length fll) (length ll)) percent-match))
                     (let loop ((inflections
                                 (unary-match (/ num1s (+ num1s num-1s))
                                              nmap))
                                (f features))
                       (if inflections
                           (loop (cdr inflections)
                                 (assert-feature-line f (first imap-elt)
                                                      (car inflections) 1))
                           (assert-feature-line f (first imap-elt) #f 1)))
                     features)))
             feature-lines mappings))

(define (unary-match ucode mappings)
  (map first (list-transform-positive mappings
               (lambda (x) (< (abs (- (second x) ucode))
                              unary-percent-match)))))

;; given the nmap, adjusts interpretations
;; for features in the nmap
;;   if inflection on featurelines
;;      get uvalue of comm-lines
;;        if more than .5xu-%match away, jump to value
;;   else
```

```
;;      test if stimulated by any comm-lines
;;        if so, randomly reselect value
(define (update-nmap nmap comm-lines feature-lines stimfeat)
  (let loop ((tnmap nmap))
    (if
     (not tnmap)
     #f
     (let* ((infl (car tnmap))
            (res (map (lambda (x)
                        (let* ((lines (map (lambda (y)
                                             (test-line comm-lines y))
                                           (second x)))
                               (numbits (length lines))
                               (num1s (- numbits (length (delq 1 lines))))
                               (num-1s (- numbits (length (delq -1 lines))))
                               (numxs (- numbits (length (delq 'x lines)))))
                          (if (= numxs 0) ;; for now, only on clean ones
                              (if (>= (/ (+ num1s num-1s) numbits)
                                      percent-match)
                                  (list (first x) (/ num1s (+ num1s num-1s)))
                                  #f)
                              'invalid)))
                      stimfeat))
            (tmap (delq #f res))
            (ifeat (list-search-positive feature-lines
                     (lambda (x) (eq? (first infl) (second x))))))
       (if (member 'invalid tmap)
           (cons infl (loop (cdr tnmap)))
       (if ifeat
           (let* ((res (list-search-positive tmap
                         (lambda (x) (eq? (first ifeat) (first x)))))
                  ;; if no value in tmap, don't change the inflection value
                  (uvalue (if res (second res) (second infl)))
                  (dif (- uvalue (second infl))))
             (if (> dif (/ unary-percent-match 2))
                 (cons (list (first infl) uvalue) (loop (cdr tnmap)))
                 (cons infl (loop (cdr tnmap)))))
           (let ((res (list-search-positive tmap
```

```
                        (lambda (x) (< (abs (- (second x) (second infl)))
                                       unary-percent-match)))))
            (if res
                (cons (list (first infl) (random 1.0))
                      (loop (cdr tnmap)))
                (cons infl (loop (cdr tnmap))))))))))

;; nmap = (list (list infl val))
;; nsquelch kills mappings which are too close together
(define (nsquelch nmap)
  (let loop ((m (random-permutation nmap)))
    (if m
        (let ((res (list-search-positive (cdr m)
                     (lambda (x) (< (abs (- (second x) (second (first m))))
                                    (* 2 unary-percent-match))))))
          (if res
              (loop (cdr m))
              (cons (car m) (loop (cdr m)))))
        #f)))

;;  Returns the Comm-codes of a symbol in an internal representation
;;  12/6: Only returns the *certain* codes
(define (find-symbol-codes imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (second elt) #f)))

;; find-symbol-uncertains: returns list of *uncertain* codes
(define (find-symbol-uncertains imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (third elt) #f)))

;; find-symbol-certainty: returns the certainty value
(define (find-symbol-certainty imap symbol)
  (let* ((smap (if imap (second imap) #f))
         (elt (list-search-positive smap (lambda (x) (eq? (first x) symbol)))))
    (if elt (fourth elt) #f)))
```

```
;; find-inflection-code: returns the unary code for a given inflection
(define (find-inflection-code imap inflection)
  (let* ((nmap (if imap (first imap) #f))
         (elt (list-search-positive nmap
                (lambda (x) (eq? (first x) inflection)))))
    (if elt (second elt) #f)))

;; Assert-line on many different lines ((line value) (line value) ...)
(define (assert-line-multiple line-list linebundle)
  (if (null? line-list)
      linebundle
      (assert-line-multiple (cdr line-list) (assert-line linebundle (caar line-lis

;; Generates Random Comm Codes
(define (random-comm-codes wps wires)
  (if (> wps 0)
      (cons (+ (random wires) 1) (random-comm-codes (- wps 1) wires))
      #f))

;; unary-signal: given a set of wires and a unary code, returns a list
;; of wires with appropriate bits on them. Signal is the %age of 1s.
(define (unary-signal wires ucode)
  (let* ((num1s (round->exact (* ucode (length wires))))
         (num-1s (- (length wires) num1s))
         (bits (random-permutation (append (make-list num1s 1)
                                           (make-list num-1s -1)))))
    (map list wires bits)))

;; Main Talk Procedure
;; 12/6: only outputs certain symbols; all newly generated randoms are certain
;; Internal-Maps is now a list of two elements. The first is inflection maps,
;; the second is symbol maps
(define (talk comm-lines feature-lines internal-maps)
  (define (loop feature-lines)
    (if (null? feature-lines)
        #f
        (let* ((cur-elmt (car feature-lines))
```

```
              (symbol (first cur-elmt))
              (inflection (second cur-elmt))
              (comm-codes (find-symbol-codes internal-maps symbol))
              (infl-code (find-inflection-code internal-maps inflection)))
         (cond ((null? comm-codes)
                (set! internal-maps
                      (set-symbol-codes
                       internal-maps symbol
                       (list (random-comm-codes num-wires-per-symbol
                                                num-wires) '() 0)))
                (loop feature-lines))
               ((null? infl-code)
                (set! internal-maps
                      (set-inflection-code internal-maps inflection
                                           (random 1.0)))
                (loop feature-lines))
               (else
                (append (unary-signal comm-codes infl-code)
                        (loop (cdr feature-lines)))))))))

(if (null? feature-lines)
    (list internal-maps comm-lines)
    (let* ((newlines (assert-line-multiple (loop feature-lines) comm-lines))
           (newmaps  internal-maps))
      (list newmaps newlines))))
```