Tel Aviv University

The Lester and Sally Entin Faculty of Humanities

Linguistics Department

Universality of Principles Tested by Iterated Learning

In fulfilment of the requirement of

0627-3777 Learning Seminar

Supervisor: Dr. Roni Katzir

Sagie Maoz

ID 021526025

sagiemao@mail.tau.ac.il

September 30, 2013

1 Introduction

A very prominent framework within modern Linguistics, in particular when dealing with Syntax, is based on the theory of *Principles and Parameters* (Chomsky and Lasnik, 1993). It suggests an underlying distinction between language attributes which are universal and between those that only appear in specific language families; the former are based on universal *principles*, while the latter are the result of the setting of *parameter* values set during the process of language acquisition.

An example for that distinction can be found when inspecting the high-level dynamics of the structure of sentences. It is understood that in every human language, the semantic value of a sentence depends on the values of its components, which is the idea behind *the Compositionality Principle of Syntactic Protection* (Webelhuth, 1992; Janssen, 1996; Cann, 1993), defined:

(1) "The syntactic properties of a complex expression are completely determined by:
a. the syntactic properties of its parts, and
b. the projection clauses."[1]

In contrast, we know that languages differ in the standard word order selected for sentences: While English, for example, mostly prefers a Subject-Verb-Object (SVO) order, Japanese is a Subject-Object-Verb (SOV) language. It has long been assumed that the word order in a particular language is the result of a combination of values of binary parameter settings such as the headedness and specifier parameters (Carnie, 1995; Ayoun, 2005; Fukui, 1993). For the purposes of this work, we will simplify this into a single multiple-choice parameter:

(2) Word Order: SVO / SOV / VSO / VOS / OSV / OVS

The idea of an attribute's universality poses questions as to its scope, its evolvement and its origins. While the Principles and Parameters theory suggests a sort of Evolutional process of human languages, others approach it more skeptically. The *Iterated Learning Model* (Smith et al., 2003; Kirby et al., 2004, 2008) offers such alternative; it shows how a population of very basic learning units, initially non-linguistic in nature, is able to transform a random language into a well-structured and easily-learned language, only by repeating their basic learning procedures enough times (iterations). The common features of the

[1] Quote taken from Webelhuth (1992, p. 40)

resulting languages can therefore be considered universal, since their formation does not depend on any initial state; they are effectively inevitable.

Kirby (2000) describes such a model for learning words and syntactical structure. The model consists of a population of initial non-linguistic individuals, who take turns trying to articulate some random meaning. If they have the proper knowledge for that meaning, they would use it, and otherwise they would either fail (with higher probability) or invent (utter a random word). The "hearing" individual would then witness the utterance and try to learn from it, employing a grammar of linear rules and a basic rule merging mechanism. Kirby reports remarkable results when running the simulation for an adequate number of iterations (5,000 in this case). After a naive first stage, where individuals made no generalisations and only acquired words directly observed, their collective strategy has shifted to merge as much rules as possible, making their grammars smaller and more efficient. Eventually, the population was able to produce stable, "minimal-length" grammars which allowed them to produce all possible meanings defined in the model. According to Kirby, such efficient minimal-length grammars employ a technique of mapping words to the smallest possible semantic values, and combining these words when trying to express a larger, more complex meaning (in this case, a sentence) – effectively implementing the Principle of Compositionality, established above (see (1)). It is therefore shown that Compositionality emerges spontaneously in languages, reinforcing the argument that it is in fact a universal principle. This, unlike word order, which differed from run to run (Kirby, p. 13), supporting that word order is indeed represented by a parameter.

The same effects were reproduced later on human learners, in Kirby et al. (2008): A test subject was presented with random words and their meanings, and were asked later to provide the words for a set of meanings (including some that were never presented). Their output was given to the next subject, simulating an iterated language learning process. Again, the resulting language quickly evolved into one with features of compositionality (in particular, compositional morphology emerged in the invented words) – yet words appeared to be random and differed between simulations.

This work attempts to re-create the simulation from Kirby (2000) by implementing the described model in computer software, and coming up with algorithms for the learning steps that were not specified in the original paper. This would allow us to make observations on the process of Iterated Learning and how it might be affected by various parameters: the choice of learning algorithms, the initial state of individuals, the setting of social dynamics in the population, and probability values. In particular, this work asks whether and how we can test the distinction8 between language attributes that are rooted in universal principles and ones that are results of different parameter values in different languages.

2 The simulation

2.1 Some terms

It would be best to define in advance some terms to be used in the description and discussion of the simulation:

Game A single run of the simulation. When run, the game initiates a population of players and runs them through the defined simulation flow. The output of a game is a stream of data describing the population's state during and at the end of the simulation.

Meaning A unit of semantic value. It it the underlying representation of ideas that can be expressed with language.

For the purpose of this work, we will limit such meanings (as did Kirby (2000)) to sets of values for *Agents*, *Predicates* and *Patients* (meaning parts). As such, the following are examples of possible meanings in the simulation:

(3) $< Agent = Zoltan, Patient = Mary, Predicate = Knows >$

(4) $< Predicate = Knows >$

Where (4) corresponds to the semantic value of the English verb *knows*, and an English sentence for (3) is *Zoltan knows Mary*.

Grammar Rule A mapping between a meaning and a text string (a word). The mapping can either be full (all meaning parts are represented) or partial (with index numbers representing missing parts), i.e.:

(5) $< Agent = Zoltan > \rightarrow zoltan$

(6) $< Agent = 1, Patient = 2, Predicate = Knows > \rightarrow 1knows2$

Grammar A set of grammar rules.

Player An individual simulating a language-using human. It has a grammar and is capable of hearing an utterance, trying to learn from it, and trying to produce an utterance for a given meaning.

2.2 The simulation flow

The fundamental workflow of the simulation described in Kirby (2000) (and presented in Figure 1) has been reproduced to follow the same steps and actions. In particular, a game cycles through a certain number of generations. In each generation, one random player is deleted and replaced with a new one, simulating a gradual turnover of the population. The new player is then the subject of an iterated learning procedure, consisting of repeated utterances by one of its two neighbours. On each such iteration, a random neighbour is chosen and attempts to articulate a random meaning to the new player.

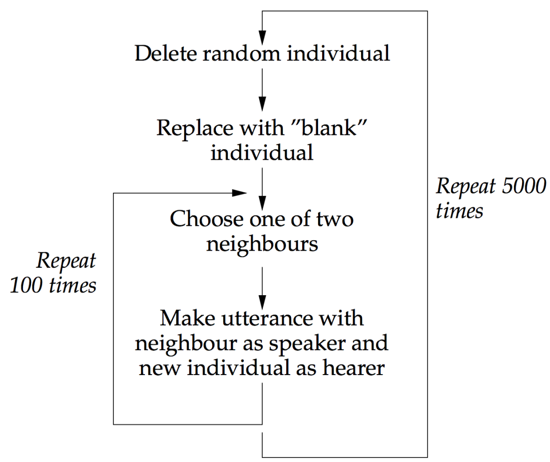

Figure 1: The main loop used in the simulations. (Kirby, 2000, p. 8)

As explained in Kirby (2000), this workflow allows the model to maintain some key features: Each player learns only from utterances produced by its neighbours; the population is ever-changing over time, replaced randomly (without any selection a-la biological evolution); and the probability that one individual will hear all forms for all the possible meanings is vanishingly small.

2.3 Model and implementation assumptions

For implementational reasons, both the original simulation and the one discussed in this work have made some assumptions regarding the model and its attributes. To clarify the features and limits of the discussed model, those are collected here.

2.3.1 Original assumptions (Kirby, 2000)

1. Sentence meanings are limited to the form $< Agent = \alpha, Patient = \beta, Predicate = \gamma >$ (p. 4).

2. The set of possible values for objects (for *Agent* and *Patient*) is *Mike, John, Mary, Tünde*, and *Zoltan*. The set of possible values for actions (for *Predicate*) is *Loves, Knows, Hates, Likes*, and *Finds*. Meanings with the same values for *Agent* as *Patient* are disallowed. Therefore, a total of 100 possible meanings is available in the model (p. 4).

3. Sentences cannot be recursive (p. 7).

4. Invention can be done partially: If a learner is missing only a part of a meaning, there is a (low) probability that they might choose to invent just that part, and embed it in the known structure (p. 6).

2.3.2 Introduced assumptions

1. Full meanings (sentences) are comprised of three parts: $Agent$, $Patient$, and $Predicate$. They are all mandatory.

2. Meaning units (whether full or partial) cannot have more than one word assigned to them.

3. Generalisations are done on the maximal possible level; a learner would always prefer the most generic rule. This specifically means that word learning is done using a Longest Common Substring matching algorithm.

4. Generalisations are done naively; similar rules are merged only when they all have an exact common word output (no word difference calculation is done, and no probabilistic negligibility is considered).

2.4 Technical details

The simulation software was written using the Ruby programming language and its standard library[2]. The software code utilises the concepts of Object Oriented Programming to represent the different model entities. The code itself is available in Appendix A, and the basic file structure behind it is described in Table 1.

File	Contents
`game.rb`	Class representing a Game, handling iterations and info collection.
`grammar.rb`	Class representing a Grammar and its lookup and learning methods.
`learner.rb`	Main (executable) file, running the simulation.
`logger.rb`	User output utility (logging results and debug information)
`meanings.rb`	Class representing a Meaning, and the code to generate all possible meanings in the model.
`player.rb`	Class representing a Player and its learning and speaking methods.
`utils.rb`	General utilities library (string matching algorithm).
`utterance.rb`	Class representing an utterance output, conceptually a tuple of word (string) and meaning.

Table 1: File structure in simulation code project.

[2]Ruby version 1.9.3 using the official interpreter (MRI): `http://ruby-lang.org/`

2.5 Algorithms

This section lists some high-level description of the algorithms used in the simulation.

2.5.1 Main flow

As mentioned, the main simulation flow is controlled by the Game class.

Algorithm 1 Game flow

```
procedure Game.Play(p)                          ▷ Play simulation on population p
    i ← 0
    while i < generations count do
        index ← random index in p
        newborn ← new Player
        p[index] ← newborn
        j ← 0
        while j < iterations count do
            speaker ← random neighbour of newborn
            invent ← probabilistic decision whether invention is possible
            m ← random meaning
            u ← speaker.speak(m, invent)
            newborn.learn(u)
            j ← j + 1
        end while
        i ← i + 1
    end while
end procedure
```

2.5.2 Meaning lookup and utterance

On each iteration, a player is requested to utter a meaning. This triggers a recursive lookup in the speaker's grammar for the meaning and any of its semantic parts.

Algorithm 2 Player speak

```
function Player.Speak(m, invent)   ▷ Speak meaning m, inventing iff invent
    word ← NULL
    if m is not empty then
        rules ← player.grammar.Lookup(meaning)
        if rules is empty then
            if invent then
                word ← random utterance
            end if
        else
            for each rule in rules do
                if rule has no variables then
                    word ← rule.word
                else              ▷ Rule has missing variables to lookup recursively
                    for each missing variable part in rule do
                        res ← player.Speak(part)
                        if res is not empty then
                            rule ← rule embedded with res
                        else      ▷ Recursive lookup failed, break loop and return
                            break
                        end if
                    end for
                    if rule has no variables then      ▷ Successful recursive lookup
                        word ← rule.word
                    end if
                end if
            end for
        end if
    end if
    return word
end function
```

2.5.3 Player learning

The hearing side of the utterance triggers a learning process that consists of two main parts: *Incorporation* (creating a new rule for the utterance and its meaning and adding it as-is to the grammar) and *Merging* (trying to compress similar rules).

Algorithm 3 Player learn

```
procedure PLAYER.LEARN(m, w)                ▷ Learn utterance w with meaning m
    rule ← New Rule "m → w"
    player.grammar.add(rule)
    player.grammar.MERGE(rule)
    player.grammar.GRAMMAR.SPLITSINGLEPARTRULES
    player.grammar.CLEAN
end procedure
```

Algorithm 4 Grammar merge

```
procedure GRAMMAR.MERGE(r)                  ▷ Merge grammar rules with r
    for each part in rule.meaning do
                          ▷ Fetch rules with exact match or missing variable in part
        rules ← all rules matching part
        if rules has more than one rule then
            word ← LONGEST COMMON SUBSTRING(words in rules)
            if word is not empty then
                for each r in rules do
                    index ← unique digit
                                                            ▷ Generalise Rule
                    r.meaning[part] ← index
                    r.word ← r.word with r.word replaced by index
                end for
                rule ← New Rule "part → word"
                grammar.add(rule)
            end if
        end if
    end for
end procedure
```

The rules compression is continued in GRAMMAR.SPLITSINGLEPARTRULES by finding rules that deal with only a single meaning part (such as (7)) and splitting them into two rules (see (8a) and (8b)). This simulates the comprehension that such rules can be fully generalised by dealing with meaning apart from structure.

(7) $< part = 1, other = 2, food = Banana > \rightarrow 1moz2$

(8) a. $< food = Banana > \rightarrow moz$

b. $< part = 1, other = 2, food = 3 \rightarrow 132 >$

Algorithm 5 Grammar split single part rules

```
procedure Grammar.SplitSinglePartRules          ▷ Split single part rules
    for each rule in grammar rules do
        if rule has only single part then
            part ← rule's known part
                                                ▷ New rule for the single-part meaning
            word ← rule.word.literal            ▷ Word sans variables
            r ← New rule "part → word"
            garmmar.add(r)
                                                ▷ Generalise original rule
            index ← unique digit
            rule.meaning[part] ← index
            rule.word ← rule.word with rule.word replaced by index
        end if
    end for
end procedure
```

One thing that was discovered while developing the simulation software, is the spontaneous creation of impossibly recursive rules, such as:

(9) $< part = 1 > \rightarrow 1a$

The generation of such rules can be easily explained when we consider the random nature of the invention of new words versus the linear nature of the merging mechanism (which looks for exact matches of substrings).

However, these rules cannot carry any useful information, and will only cause the lookup recursive in PLAYER.SPEAK to enter an endless loop. Therefore, such rules are searched for and deleted in GRAMMAR.CLEAN.

Algorithm 6 Grammar cleaning

```
procedure GRAMMAR.CLEAN                              ▷ Clean grammar
    for each rule in grammar rules do
        if rule has a single part part then
            if rule[part] is a variable then
                delete rule
            end if
        end if
    end for
end procedure
```

2.6 Differences from Kirby 1998

Kirby (1998) describes a similar simulation for an Iterated Learning Model, with goals and results corresponding to the high-level description in Kirby (2000). However, that simulation differs from the implementation discussed in this work, in the following ways:

1. Kirby (1998) simulates noise by a 1:1000 probability of the speaker transmitting a random utterance (regardless of target meaning or the speaker's knowledge). However, trying to generate noise similarly for this work showed no significant effect on the results.

2. When a speaker lacks grammar rules for some meaning, they sometimes generate a random utterance (in both simulations), and that utterance is learned by the hearer. In Kirby (1998), the speaker also goes through such learning. For implementational reasons, this was omitted, considering that such process would probably have little effect (since grammar growth within a generation is measured mostly on the hearing player side).

3. In Kirby (1998), every grammar started out with rules for each of the alphabet terminals. This is described as a technical requirement, which was not required for this simulation. This also has the effect (in Kirby (1998, 2000)) of players having a larger number of grammar rules than of possible meanings in the first stage of the simulation.

4. The Kirby (1998) simulation handled nouns (*Object*s) once, allowing them to appear both as *Patient*s and *Agent*s. This was not reproduced in this work, which treated, for example, $< Patient = Mary >$ as a completely different entity than $< Agent = Mary >$.

3 Results

The simulation was run several times using a default set of parameters, aimed at reproducing the Kirby (2000) tests: A population size of 10 players, invention probability of 1:50, over 5,000 generations with 100 learning iterations per generation. Population averages of grammar size[3] and meanings count[4] were measured at the end of each generation.

All of these runs showed consistent results in relation to one another and compared to the expectations detailed in Section 1 (Introduction). Notably, the simulated population went through a process of first mapping words to full meanings (sentences), and then gradually a compositional structure emerged in their grammar. After a certain amount of generations (roughly around 4,000 generations), the average grammar evolved into a minimal-length grammar and meanings count was virtually 100% of possible meanings.

3.1 Results with different parameters

Since the software was designed to allow users to change the value of most numeric parameters, it was easy to test the difference in results for different values of these parameters. Testing the results after changing each value reaffirmed the simulation's relation with its model:

Population size Changing the size of the population caused little surprises: Smaller populations had more trouble to learn from each other, and it took more generations to achieve an efficient minimal-length grammar. Similarly, increasing the population size allowed for an expedited learning process.

Number of learning iterations As to be expected, increasing the number of iterations a new player is exposed to utterances helped this player to perfect its grammar, resulting in a quicker learning process.

[3] Number of rules in a player's grammar.

[4] Number of meanings (of those possible in the model) that a player can express without invention.

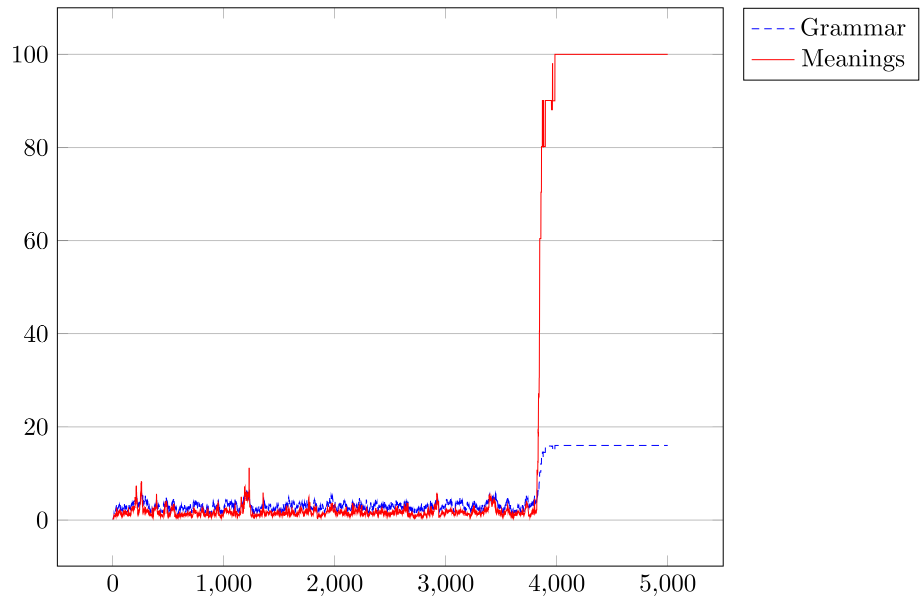

Figure 2: Population average of size and meanings over a typical run of the simulation. Note that circa the 4,000th generation, meanings count climbs to 100, while grammar size stabilises around 16.

Probability of invention Increased likelihood of invention means significantly more data to be processed in the early stages (when players have no linguistic knowledge). However, this also creates noise in the form of words that cannot be merged. That is why changing the probability value did little to affect the results of the simulation.

4 Discussion

As mentioned and can be seen in Figure 2, all runs of the simulation were alike in their final outcome, in that they involved minimal-length grammars that relies on compositionality to allow the speaker to express a wide selection of semantic values, including those that it was not exposed to. For example, while players in the first stage of the simulation (before the emergence of such minimal grammars) used a grammar like that in (10), the 'smarter', more evolved players, used a compositional grammar that is much more expressive, such as the one in (11).

(10) a. $< Agent = Mike, Patient = Mary, Predicate = Finds > \rightarrow$ *dbacd*

b. $< Agent = Mike, Patient = Mary, Predicate = Likes > \rightarrow$ *dbbbcd*

c. $< Agent = Mike, Patient = John, Predicate = Likes > \rightarrow$ *cbbbcd*

d. $< Agent = Mike, Patient = John, Predicate = Finds > \rightarrow$ *cbacd*

e. $< Agent = Zoltan, Patient = Mary, Predicate = Finds > \rightarrow$ *dbaddd*

f. $< Agent = Zoltan, Patient = Mary, Predicate = Likes > \rightarrow$ *dbbbddd*

(11) a. $< Agent = 1, Patient = 2, Predicate = 3 > \rightarrow 231$

b. $< Agent = Mike > \rightarrow$ *cd*

c. $< Agent = Zoltan > \rightarrow$ *ddd*

d. $< Predicate = Likes > \rightarrow$ *b*

e. $< Predicate = Finds > \rightarrow$ *a*

f. $< Patient = Mary > \rightarrow$ *db*

g. $< Patient = John > \rightarrow$ *cb*

Such behaviour is consistent with the one described in Kirby (2000), and reaffirms the claim that indeed, the Compositionality Principle of Syntactic Protection (defined in (1)) emerges spontaneously in languages, and therefore can be considered as universal.

In contrast, when inspecting the word order property of the emerging languages, it is clear that such property is not at all persistent. In the simulation results, the word order is defined by a single grammar rule similar to (11a). The order of the index numbers (which correspond to semantic parts) ultimately sets the order of elements in a full sentence. The resulting grammars in each simulation run were indeed different in that particular rule. This is presumably due to the fact that the first newly introduced words, which are random strings, represent full sentences (as seen in (10)), and those are later broken down to parts by Grammar.Merge. This heterogeneous nature of emerging word order in the simulation results matches our hypothesis and confirms that word order is not rooted in the basic nature of language or its learning process. In fact it depends on the environment in which the learning is done, which affects the first invented words and the quality of rule merging.

In summary, simulating language learning using an Iterated Learning model allows us to distinguish between language features which are inevitable and those that are local to a specific environment and are subject to change. We have used this observation to reaffirm the universality of compositionality versus the locality of word order, as assumed by the theory of Principles and Parameters.

Further research can and should be done studying other language features and behaviours. Hopefully the simulation software and algorithms can be used to verify the universality of other linguistic features; perhaps additional syntactic analysis tools would allow the inspection of the principles of the Government and Binding theory (Chomsky, 1993). It might also be interesting (although probably more challenging) to simulate the development of phonological features and study the emerging order of constraints assumed in the Optimality Theory (Prince and Smolensky, 2008). Additionally, it will be interesting to introduce to the simulation further variables representing the environment, and experiment with which of these control which parameters.

References

Ayoun, D. (2005). *Parameter Setting in Language Acquisition.* Continuum, London.

Cann, R. (1993). *Formal semantics: an introduction.* Cambridge University Press.

Carnie, A. H. (1995). *Non-verbal predication and head-movement.* PhD thesis, Massachusetts Institute OF Technology.

Chomsky, N. (1993). *Lectures on Government and Binding: The Pisa Lectures.* Studies in generative grammar. Bod Third Party Titles.

Chomsky, N. and Lasnik, H. (1993). The theory of principles and parameters. *Syntax: An international handbook of contemporary research*, 1:506–569.

Fukui, N. (1993). Parameters and optionality. *Linguistic Inquiry*, pages 399–420.

Janssen, T. M. (1996). Compositionality.

Kirby, S. (1998). Language evolution without natural selection: From vocabulary to syntax in a population of learners. *Edinburgh Occasional Paper in Linguistics EOPL-98-1*.

Kirby, S. (2000). Syntax without Natural Selection: How compositionality emerges from vocabulary in a population of learners. In Knight, C., editor, *The Evolutionary Emergence of Language: Social Function and the Origins of Linguistic Form*, pages 303–323. Cambridge University Press.

Kirby, S., Cornish, H., and Smith, K. (2008). Cumulative cultural evolution in the laboratory: An experimental approach to the origins of structure in human language. *Proceedings of the National Academy of Sciences*, 105(31):10681–10686.

Kirby, S., Smith, K., and Brighton, H. (2004). From UG to universals: Linguistic adaptation through iterated learning. *Studies in Language*, 28(3):587–607.

Prince, A. and Smolensky, P. (2008). *Optimality Theory: Constraint interaction in generative grammar*. Wiley. com.

Smith, K., Kirby, S., and Brighton, H. (2003). Iterated learning: A framework for the emergence of language. *Artificial Life*, 9(4):371–386.

Webelhuth, G. (1992). *Principles and parameters of syntactic saturation*. Oxford University Press.

Appendix A Simulation source code

Listing 1: code/game.rb

```
require './logger'
require './player'

class Game
  attr_accessor :population

  def initialize(options)
    @options = options

    @generation = 0

    init_population(@options[:population])
  end

  def play(generations=nil, iterations = nil)
    generations ||= @options[:generations]

    generations.times do
      @generation+= 1
      play_step(iterations)
    end

    MyLogger.debug "Population: #{population}"
  end

  def grammars
    population.map do |player|
      player.grammar
    end
  end

  private

  def init_population(size)
    self.population = []
    size.times do
      spawn_player
    end
  end

  def play_step(iterations = nil)
    iterations ||= @options[:iterations]
```

```

    # Replace a random player
    index = random_player_index
    population[index] = Player.new(@options[:probability])

    iterations.times do |i|
      speaker   = population[random_neighbor_index(index)]
      utterance = speaker.speak(Meanings.sample)
      if utterance # something was said
        population[index].learn(utterance)
      end

      log_info(i) if @options[:print_after] == :iteration
    end

    log_info if @options[:print_after] == :generation

    population.each do |player|
      player.age+= 1
    end
  end

  def random_player_index
    rand(population.size)
  end

  def random_neighbor_index(index)
    direction = [+1, -1].sample
    (index + direction) % population.size
  end

  def spawn_player
    population << Player.new(@options[:probability])
  end

  def average_grammar_size
    sizes = grammars.map(&:size)
    sizes.inject(:+).to_f / population.size
  end

  def average_meaning_count
    sizes = population.map(&:meaning_count)
    sizes.inject(:+).to_f / population.size
  end

  def log_info(iteration = nil)
```

```
    info = []
    info << "g#%4d" % @generation
    (info << "i#%3d" % iteration) if iteration
    if @options[:print_grammar_size]
      info << "grammar: %5.1f" % average_grammar_size
    end
    if @options[:print_meaning_count]
      info << "meanings: %5.1f" % average_meaning_count
    end
    MyLogger.info info.join("\t")
  end
end
```

Listing 2: code/grammar.rb

```
require './utils'
require './meanings'

class Grammar < Hash
  class Rule
    attr_accessor :meaning, :word

    def initialize(meaning, word)
      self.meaning = meaning.clone
      self.word    = word.clone

      @_last_index = 0
    end

    # generalise part with a new index
    def generalise_part!(part, new_word)
      index = generate_index
      meaning[part] = index
      word.sub! new_word, index.to_s
    end

    # embed new_word in part, replacing index
    def embed!(part, index, new_word)
      self.meaning[part] = :embedded
      self.word = word.sub(index.to_s, new_word)
    end

    # literal (non-digits) part of word
    def literal
      word.gsub(/[0-9]/, '')
    end

```

```
    # deep clone
    def clone
      super.tap do |rule|
        rule.meaning = self.meaning.clone
      end
    end

    def to_s
      "#{meaning} -> '#{word}'"
    end

    private
    def generate_index
      @_last_index += 1
    end
  end

  # learn a new rule
  def learn(meaning, word=nil)
    rule = nil

    if meaning.is_a? Rule
      rule = meaning
    elsif word
      rule = Rule.new(meaning, word)
    end

    add_rule(rule) unless rule.nil?
  end

  # find all rules with same part=meaning
  def with(part, meaning)
    values.select do |rule|
      rule.meaning[part] == meaning
    end
  end

  # merge parts of a given rule
  def merge(rule)
    rule.meaning.each do |part, meaning|
      if rule.meaning.has?(part)
        new_rule = merge_part(rule.meaning[part], part)
        learn(new_rule) unless new_rule.nil?
      end
    end
  end
```

```

  def clean!
    new_rules = []

    each do |key, rule|
      # split single-part rules
      if rule.meaning.single_part?
        new_rules << split_single_rule(rule)
      end

      # remove unrealistic recursive rules ”1 -¿ 1a”
      if rule.meaning.known_parts.count == 0
        if rule.meaning.unknown_parts.count <= 1
          delete_rule rule
        end
      end
    end

    new_rules.each do |rule|
      learn rule
    end
  end

  # find all rules matching a meaning
  def lookup(target)
    select do |key, rule|
      target.matches?(rule.meaning)
    end.values
  end

  private

  # add a rule to grammar
  def add_rule rule
    self[rule.meaning.to_sym] = rule
  end

  # remove a rule from grammar
  def delete_rule rule
    delete rule.meaning.to_sym
  end

  # merge all rules with same part=meaning
  def merge_part(meaning, part)
    rules = with(part, meaning)

```

```
    if rules.count > 1
      words = rules.map { |r| r.word }
      new_word = Utils.longest_common_substr words, /[0-9]/

      unless new_word.empty?
        # generalise that part in all rules
        rules.each do |r|
          delete_rule(r)
          r.generalise_part! part, new_word
          add_rule(r)
        end

        # create new rule for that part=meaning
        new_meaning = Meaning.new
        new_meaning[part] = meaning
        Rule.new(new_meaning, new_word)
      end
    end
  end

  # split a single-part rule
  #   part=1, other=2, food=Banana -¿ 1moz2
  # into two rules: one for meaning, one for structure
  #   food=Banana -¿ 1moz2
  #   part=1, other=2, food=3 -¿ 132
  def split_single_rule rule
    # rule has a single part
    part = rule.meaning.known_parts.first

    # new rule for the single meaning
    new_word = rule.literal # ('moz')
    new_meaning = Meaning.new # (food=Banana)
    new_meaning[part] = rule.meaning[part]
    new_rule = Rule.new(new_meaning, new_word) # (food=Banana -¿ moz)
    add_rule(new_rule)

    # generalize the original rule
    #  (part=1, other=2, food=3 -¿ 132)
    rule.generalise_part! part, new_word
    rule
  end
end
```

Listing 3: code/learner.rb

```
#!/usr/bin/env ruby

```

```
require './logger'
require './game'
require './grammar'
require './meanings'

require 'optparse'
options = {
  :population => 10,
  :generations => 5000,
  :iterations => 100,
  :probability => 0.02,
  :print_grammars => false,
  :print_after => :generation,
  :print_grammar_size => true,
  :print_meaning_count => false,
}
OptionParser.new do |opts|
  opts.banner = "Usage:␣learner.rb␣[options]"

  opts.on("-p␣N", "--population␣N", Integer,
          "Set␣population␣size") do |v|
    options[:population] = v
  end

  opts.on("-g␣N", "--generations␣N", Integer,
          "Set␣generations␣count") do |v|
    options[:generations] = v
  end

  opts.on("-i␣N", "--iterations␣N", Integer,
         "Set␣iterations␣count␣(for␣each␣generation)") do |v|
    options[:iterations] = v
  end

  opts.on("--probability␣N", Float,
         "Set␣invention␣probability") do |p|
    options[:probability] = p
  end

  opts.on("-d", "--debug",
           "Show␣debug␣messages") do |debug|
    require 'pry'
    options[:debug] = true
    MyLogger.level = Logger::DEBUG
  end

```

```
  opts.on("--[no-]print-grammar-size",
          "Print␣grammar␣sizes␣on␣each␣info␣log") do |v|
    options[:print_grammar_size] = v
  end

  opts.on("--[no-]print-meaning-count",
          "Print␣meaning␣counts␣on␣each␣info␣log") do |v|
    options[:print_meaning_count] = v
  end

  opts.on("--print-after␣[iteration|geneation]",
         "Set␣info␣log␣timing") do |v|
    options[:print_after] = :iteration if v == 'iteration'
  end

  opts.on("--print-grammars",
         "Print␣final␣grammars") do |print_grammars|
    options[:print_grammars] = true
  end
end.parse!

game = Game.new(options)
game.play

if options[:print_grammars]
  puts game.grammars
end

if options[:debug]
  binding.pry
end
```

Listing 4: code/logger.rb

```
require 'logger' # Ruby's Logger

MyLogger = Logger.new(STDOUT)

MyLogger.formatter = proc do |severity, datetime, progname, msg|
  "[#{severity}]␣#{msg}\n"
end

MyLogger.level = Logger::INFO
```

Listing 5: code/meanings.rb

```
# encoding: UTF-8
```

```ruby

class Meaning
  Categories = [:agent, :predicate, :patient]

  Categories.each do |cat|
    attr_accessor cat
  end

  def initialize(agent = nil, predicate = nil, patient = nil)
    self.agent     = agent
    self.predicate = predicate
    self.patient   = patient
  end

  def values
    {
      :agent => agent,
      :predicate => predicate,
      :patient => patient,
    }
  end

  def [](part)
    values[part.to_sym]
  end

  def []=(part, value)
    send("#{part}=", value) if Categories.include? part
  end

  def each(&block)
    values.each(&block)
  end

  def has?(part)
    !values[part].nil?
  end

  def missing?(part)
    values[part].is_a? Numeric
  end

  def matches?(other)
    values.keys.inject(true) do |mem, key|
      mem && matches_part?(other, key)
    end
```

```
  end

  def full?
    !empty? && missing_parts.count == 0
  end

  def partial?
    !empty? && missing_parts.count > 0
  end

  def empty?
    values.keys.inject(true) do |res, part|
      res && !has?(part)
    end
  end

  def missing_parts
    values.keys.inject({}) do |res, part|
      res[values[part]] = part if missing?(part)
      res
    end
  end

  def known_parts
    values.keys.inject([]) do |res, part|
      res << part if has?(part) && !missing?(part)
      res
    end
  end

  def unknown_parts
    values.keys.inject([]) do |res, part|
      res << part if has?(part) && missing?(part)
      res
    end
  end

  def single_part?
    partial? && known_parts.count == 1
  end

  def to_s(include_missing = true)
    values.keys.inject([]) do |res, part|
      value = values[part]
      unless value.nil?
        res << "#{part}=#{value}" if include_missing || !missing?(part)
```

```
      end
      res
    end.join(',')
  end

  def to_sym
    to_s(false).to_sym
  end

  private
  def matches_part?(other, part)
    (has?(part) && other.missing?(part)) ||
      other[part] == self[part]
  end
end

MeaningObjects = [
  :Mike, :John, :Mary, :'Tu␣ÌĹnde', :Zoltan
]

MeaningActions = [
  :Loves, :Knows, :Hates, :Likes, :Finds
]

Meanings = []

MeaningObjects.each do |agent|
  MeaningActions.each do |predicate|
    (MeaningObjects - [agent]).each do |patient|
      Meanings << Meaning.new(agent, predicate, patient)
    end
  end
end
```

Listing 6: code/player.rb

```
require './logger'
require './grammar'
require './meanings'
require './utterance'

class Player
  attr_accessor :id
  attr_accessor :age
  attr_accessor :grammar

  Alphabet = [ 'a', 'b', 'c', 'd' ]
```

```

  def initialize(probability)
    self.id = Player.generate_id
    self.grammar = Grammar.new
    self.age = 0

    @probability = probability
  end

  # (try to) articulate a given meaning
  def speak meaning
    MyLogger.debug "Player ##{id} speaking #{meaning}"
    word = lookup(meaning, should_invent?)
    Utterance.new(meaning, word) unless word.nil?
  end

  # learn from a neighbour's utterance
  def learn utterance
    MyLogger.debug "Player ##{id} learning #{utterance}"
    # 1. Incorporation
    rule = grammar.learn utterance.meaning, utterance.word
    # 2. Merging
    grammar.merge rule if rule
    # 3. Cleaning
    grammar.clean!
  end

  # count possible meanings available
  def meaning_count
    Meanings.inject(0) do |count, m|
      count+= 1 if can_speak?(m)
      count
    end
  end

  def to_s
    "<Player ##{id} age:#{age} " +
    "grammar.size:#{grammar.count}>"
  end

  def self.generate_id
    @_last_id ||= 0
    @_last_id+= 1
  end

  private
```

```

  # whether to invent a new word
  def should_invent?
    rand(100) < @probability * 100
  end

  # utter a random word
  def utter_randomly
    length = Utterance::MinLength +
      rand(Utterance::MaxLength - Utterance::MinLength)
    (0...length).map{ Alphabet.sample }.join
  end

  # is meaning possible thru grammar
  def can_speak?(meaning)
    !lookup(meaning, false).nil?
  end

  # return word representing meaning, if available
  def lookup(meaning, should_invent=false)
    word = nil
    unless meaning.empty?
      rules = grammar.lookup(meaning)
      if rules.empty?
        if should_invent
          word = utter_randomly
          # self.learn Utterance.new meaning, word
        end
      else
        rules.sort_by! do |rule|
          rule.meaning.missing_parts.count
        end.reverse!
        rules.each do |rule|
          if rule.meaning.full?
            word = rule.word
            break
          else
            current = rule.clone
            current.meaning.missing_parts.each do |index, part|
              required = Meaning.new
              required[part] = meaning[part]
              res = lookup(required, should_invent)
              if res.nil?
                word = nil
                break
              else
```

```
                current.embed!(part, index, res)
              end
            end
            if current.meaning.full?
              word = current.word
              break
            end
          end
        end
      end
    end
    word
  end
end
```

Listing 7: code/utils.rb

```
class Utils
  # adopted from http://stackoverflow.com/a/2158481/107085
  def self.longest_common_substr(strings, disallow = nil)
    shortest = strings.min_by(&:length)
    maxlen = shortest.length
    maxlen.downto(0) do |len|
      0.upto(maxlen - len) do |start|
        substr = shortest[start, len]
        if disallow.nil? || (substr =~ disallow) == nil
          return substr if strings.all? { |str| str.include? substr }
        end
      end
    end
  end
end
```

Listing 8: code/utterance.rb

```
require './meanings.rb'

class Utterance
  MinLength = Meaning::Categories.length
  MaxLength = 8

  attr_accessor :meaning
  attr_accessor :word

  def initialize(meaning, word)
    self.meaning = meaning
    self.word = word
```

```
  end

  def to_s
    "'#{word}'␣(#{meaning})"
  end
end
```