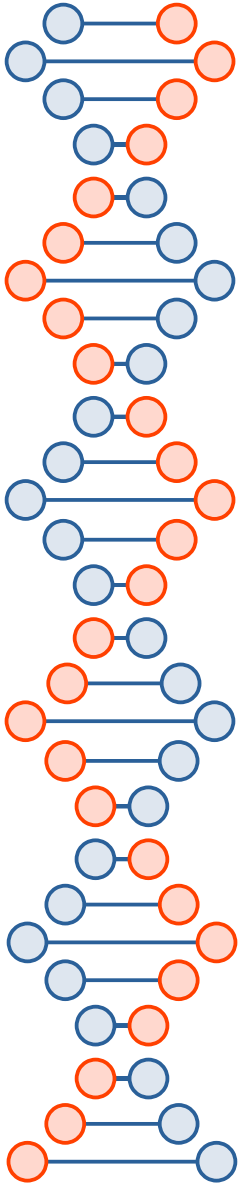


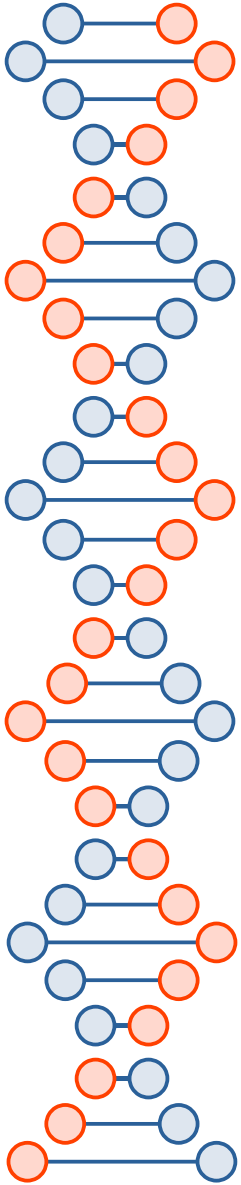
GEMS System: Tutorial

Presented at AI-2023 Workshop

"Computational Discovery in Social Sciences"

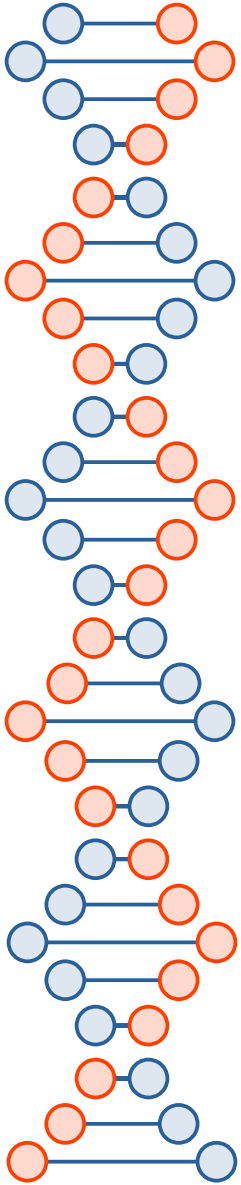
by Dr. Peter Lane, University of Hertfordshire

12th December 2023

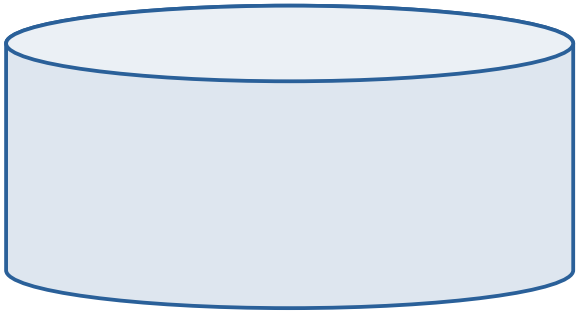


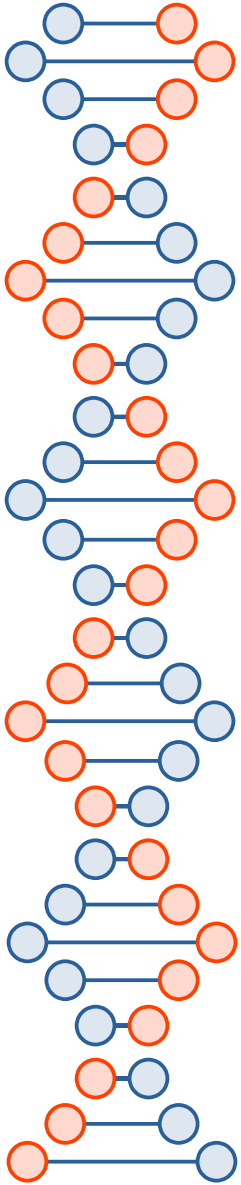
Overview

- setting up task definitions for scientific experiments
- defining a search space of candidate models
- searching techniques, such as Genetic Programming
- visualisation and analysis of results

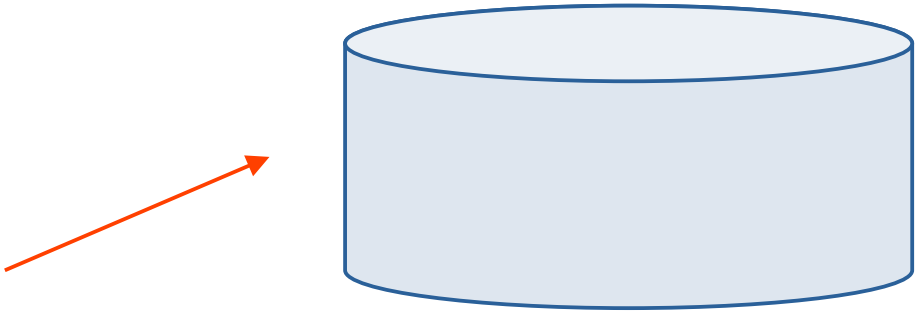


Part 1: Creating Models

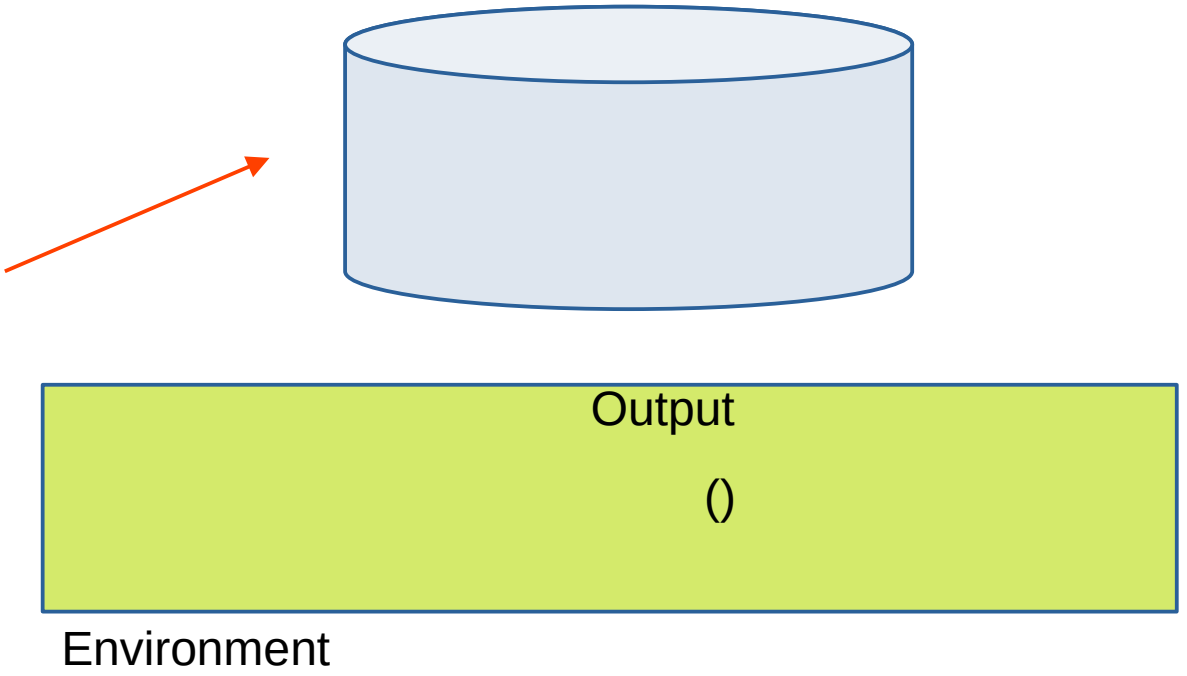


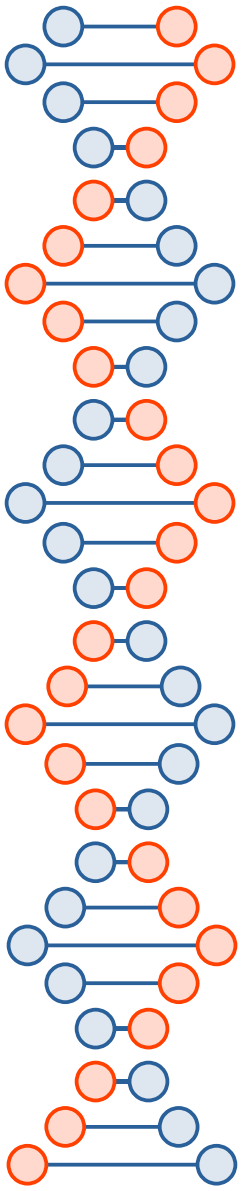


Part 1: Creating Models

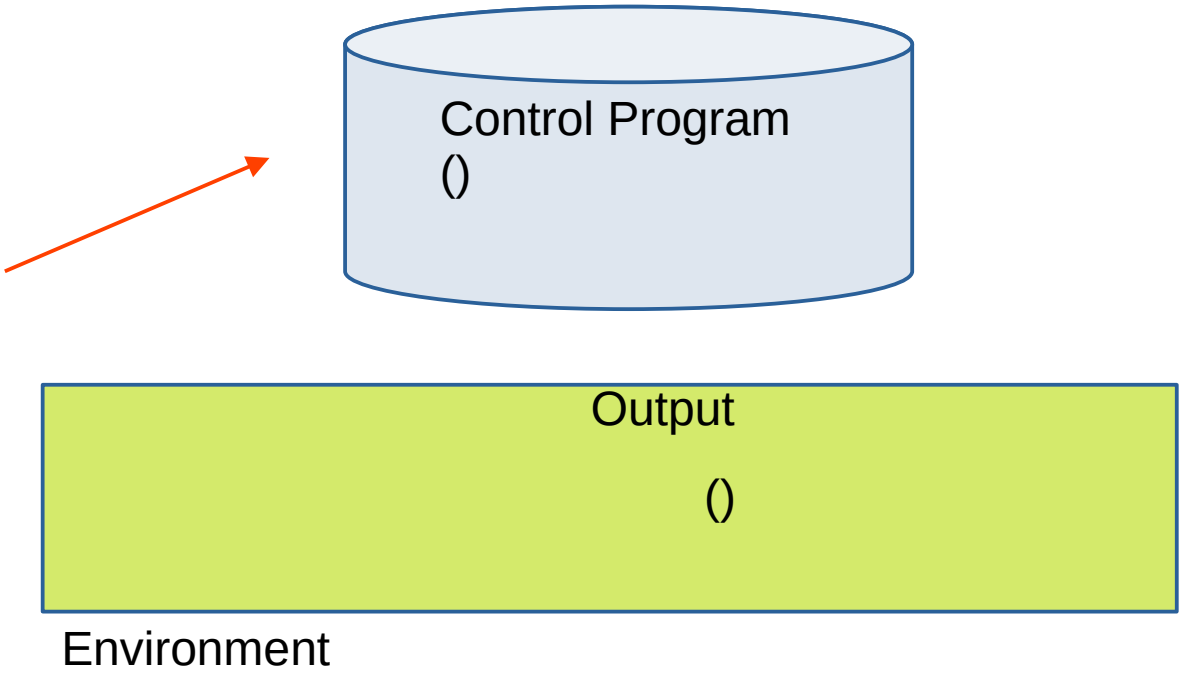


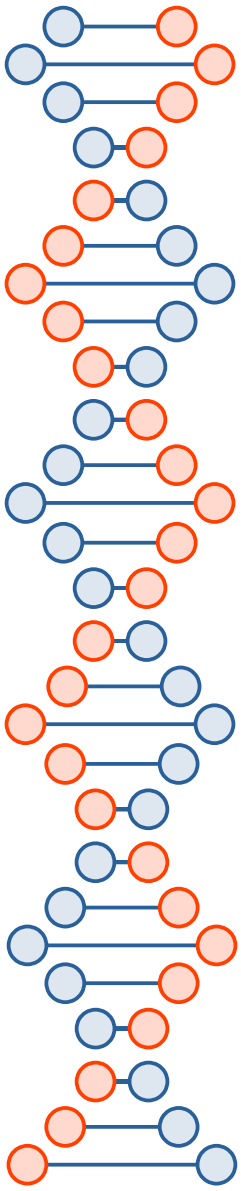
Part 1: Creating Models



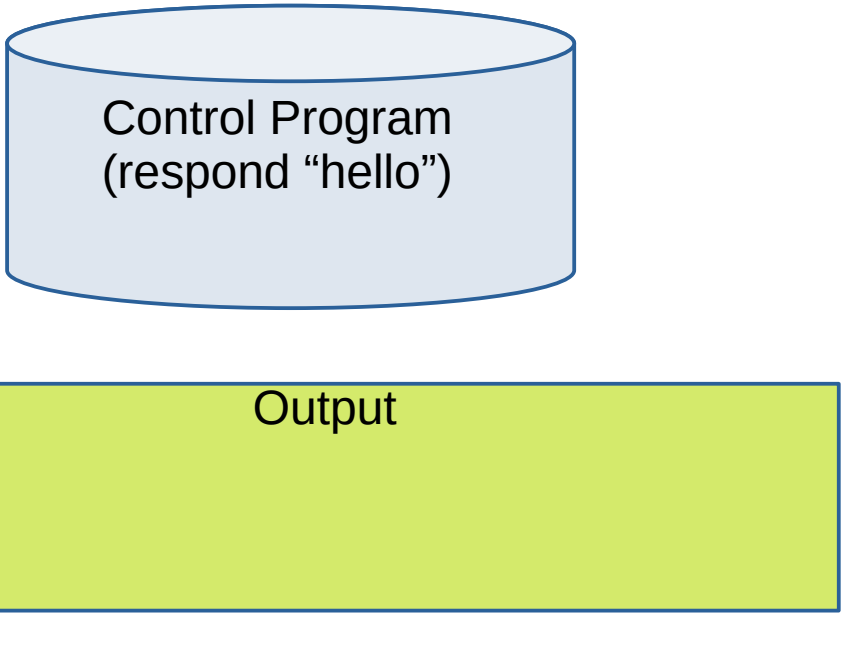


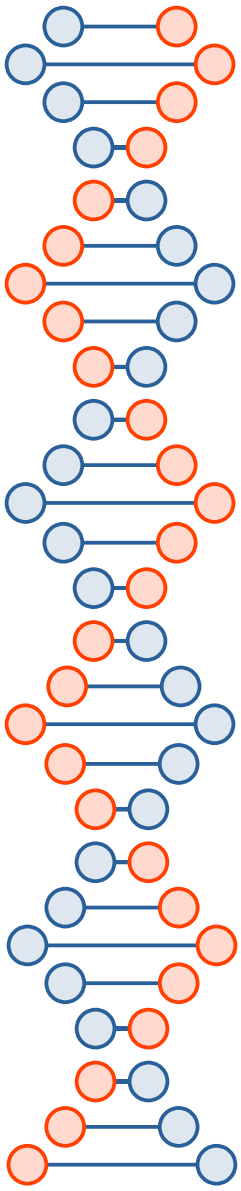
Part 1: Creating Models



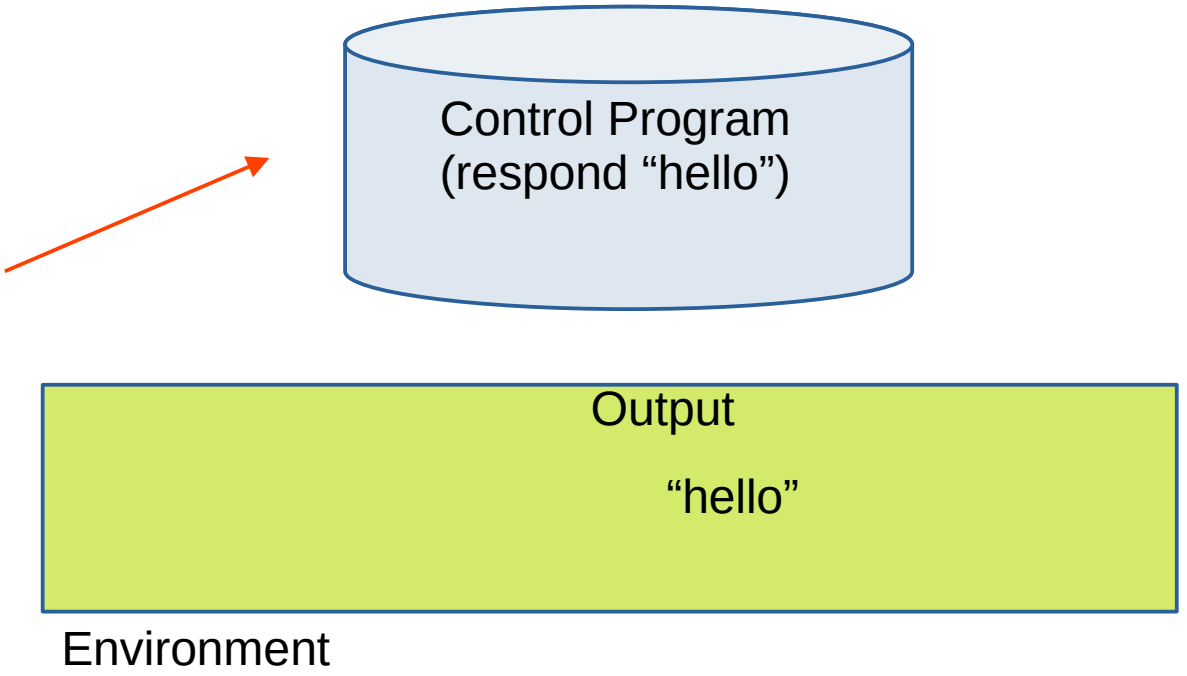


Part 1: Creating Models

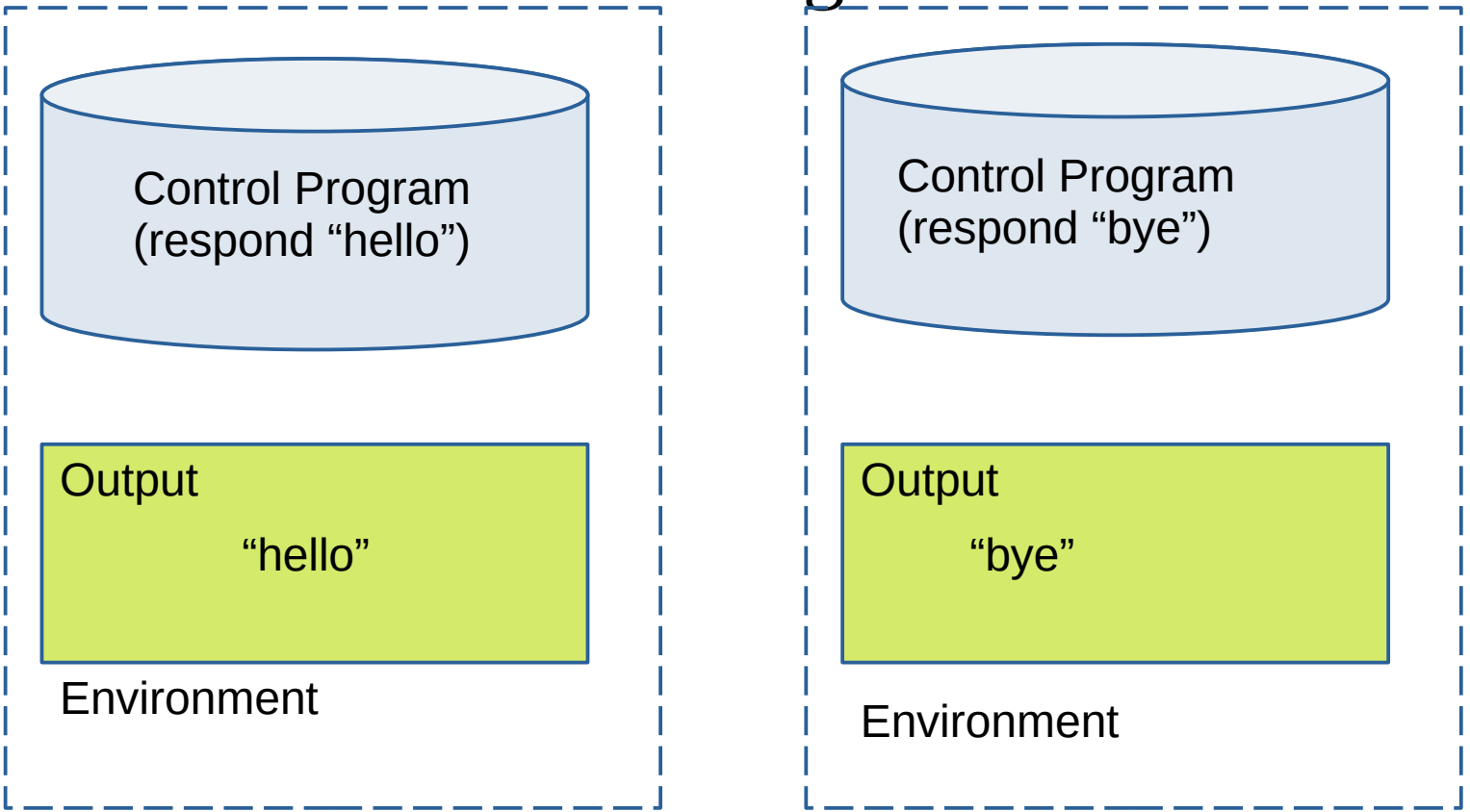




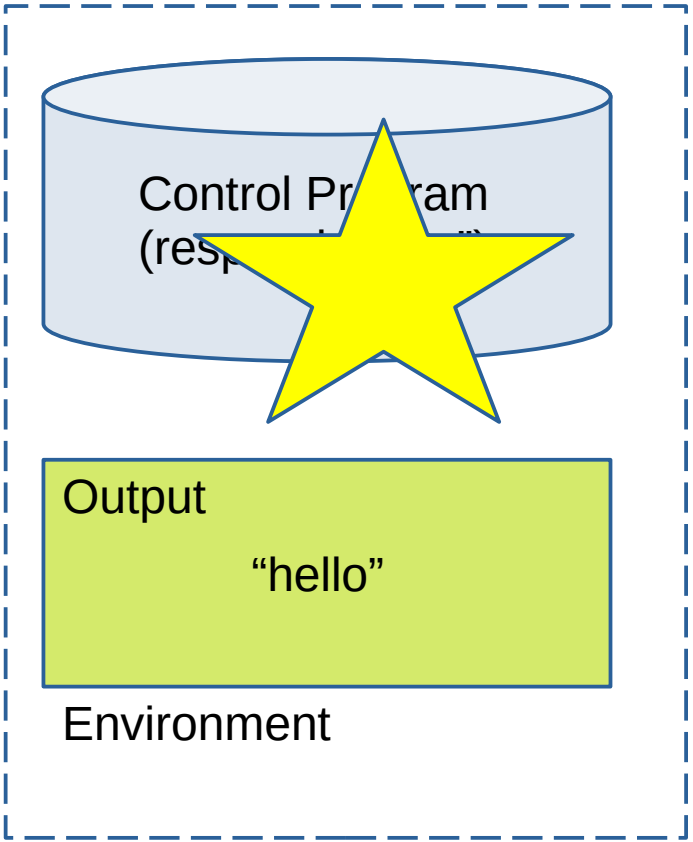
Part 1: Creating Models

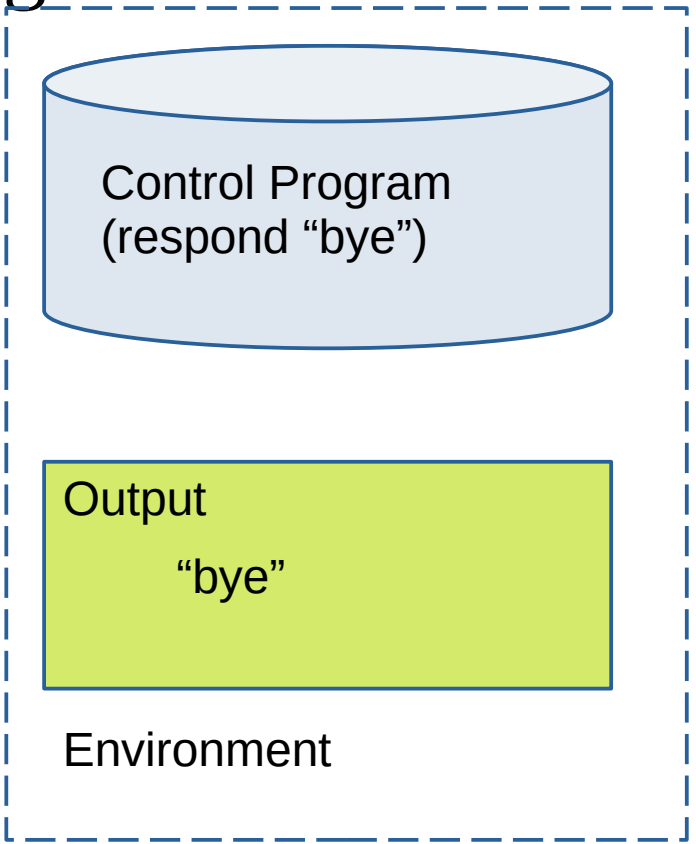


Part 1: Creating Models

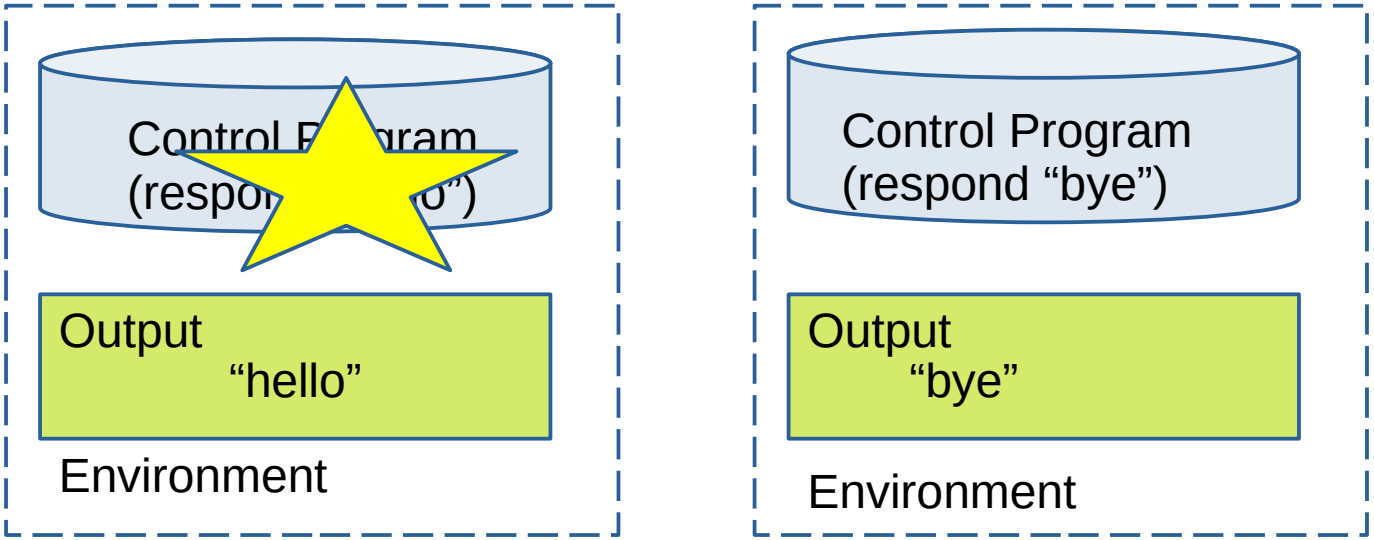


Part 1: Creating Models





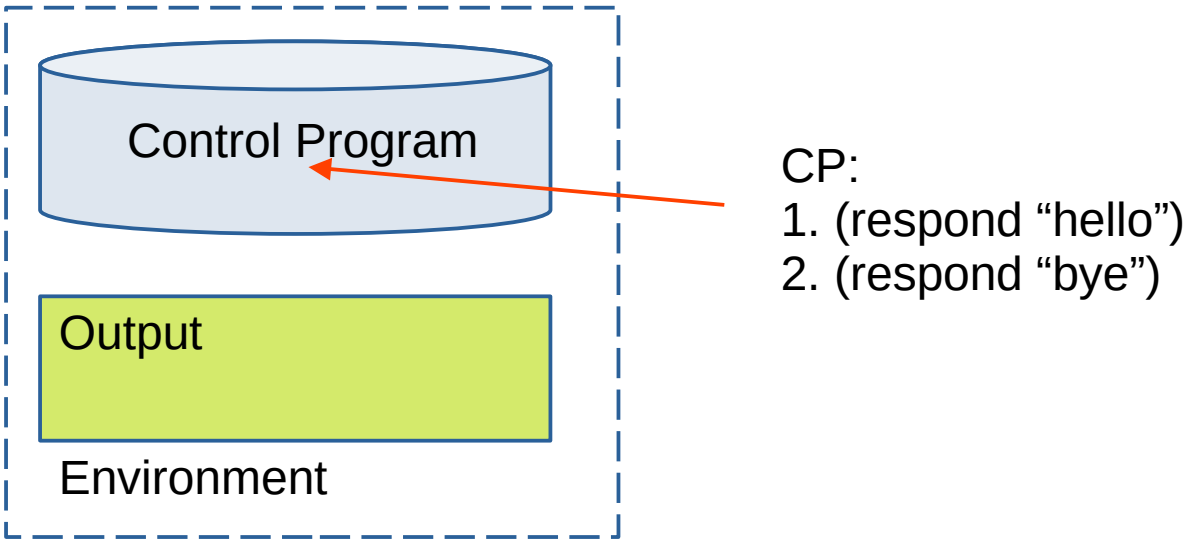
Part 1: Creating Models



Evaluation Function:

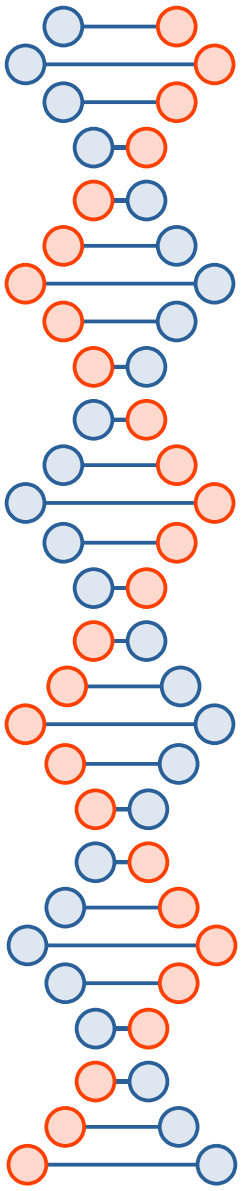
1. run model
2. observe output
3. if output is “hello” then model is correct, else incorrect

Part 1: Creating Models

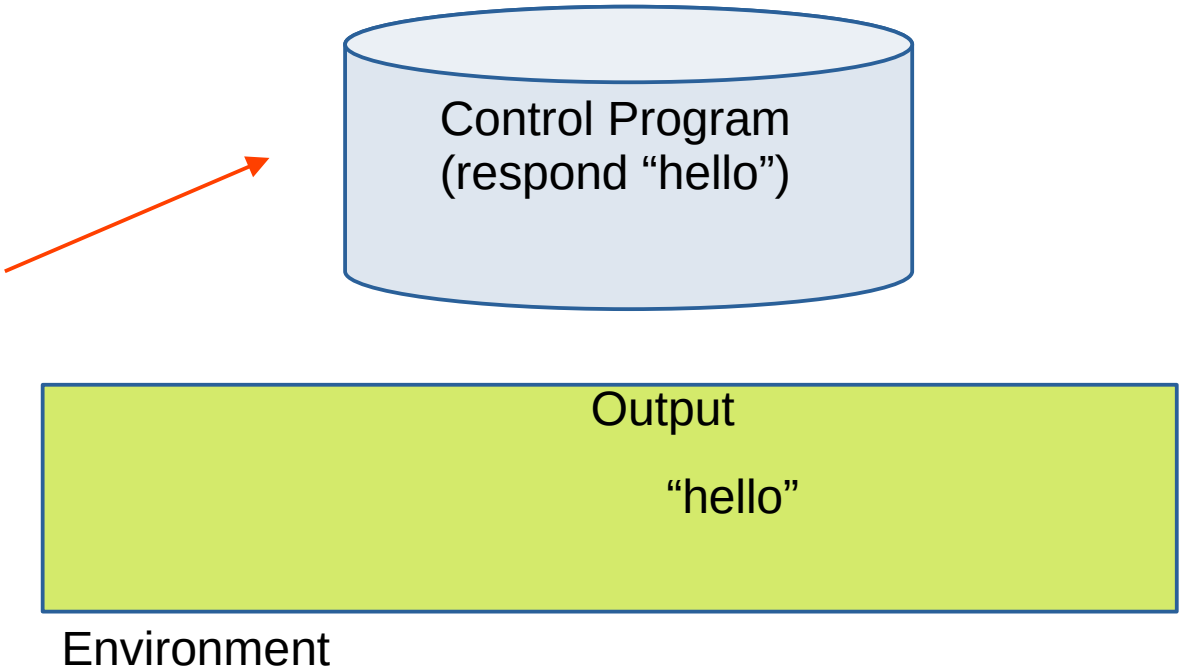


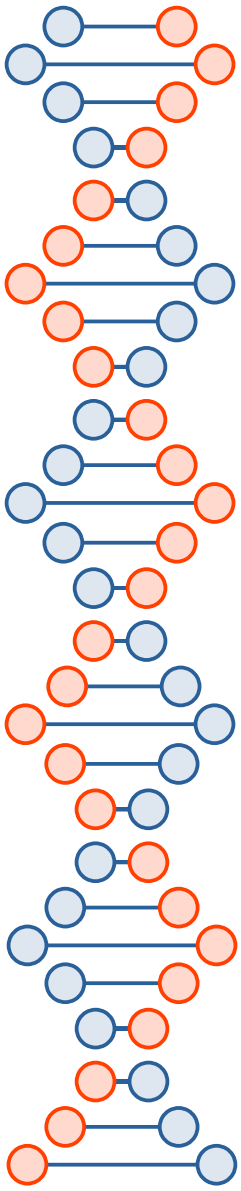
What does (respond “hello”) do?

(respond TEXT) places the TEXT into the Output slot of Environment

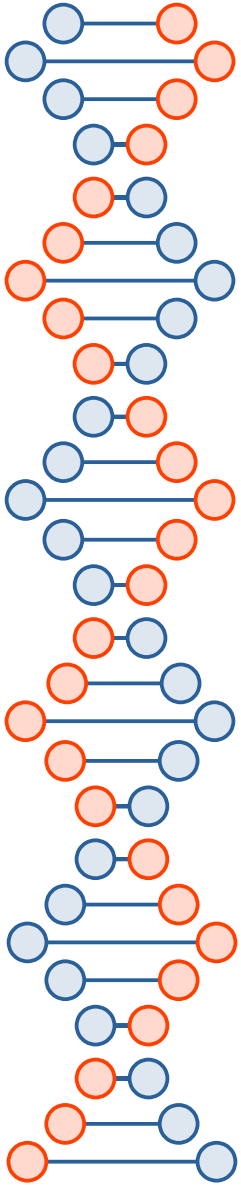


Part 1: Creating Models

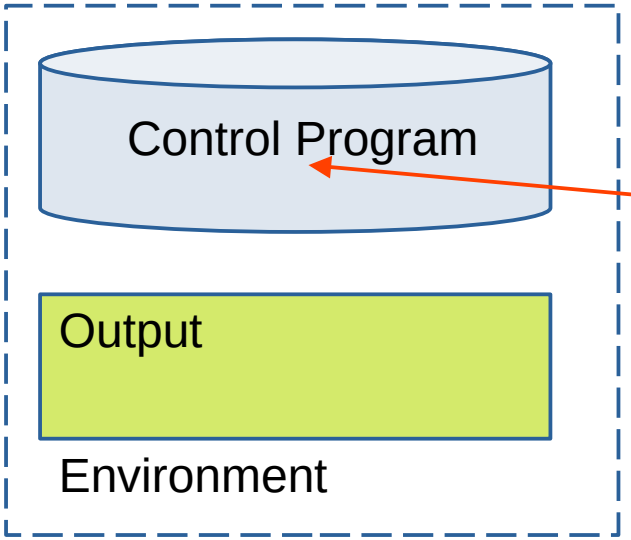




```
(defstruct model
  response ; hold response from model
  )

(defun evaluate-program (program)
  (let ((result (run-experiment program)))
    (if (equalp "hello" result) 0 1) ; compute fitness
    ))

(defun run-experiment (program)
  (let ((md (make-model :response "")))
    (interpret program md)
    (model-response md)))

(defun interpret (operator md)
  (case (operator-label operator)
    (:respond-hello
      (setf (model-response md) "hello"))
    (:respond-bye
      (setf (model-response md) "bye"))
    (otherwise
      (error "interpret: unknown operator ~a" (operator-label operator)))))
```



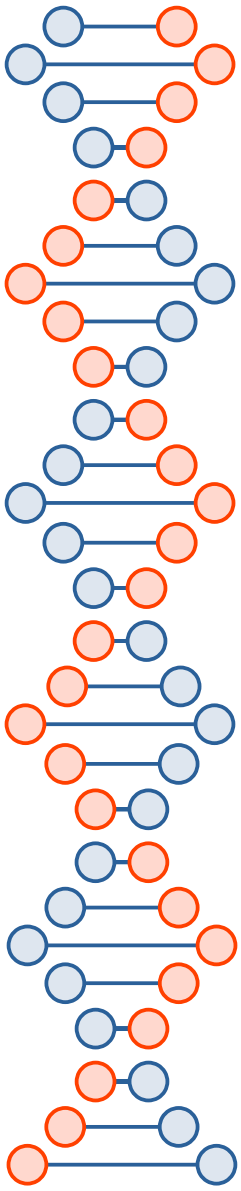
Part 1: Creating Models



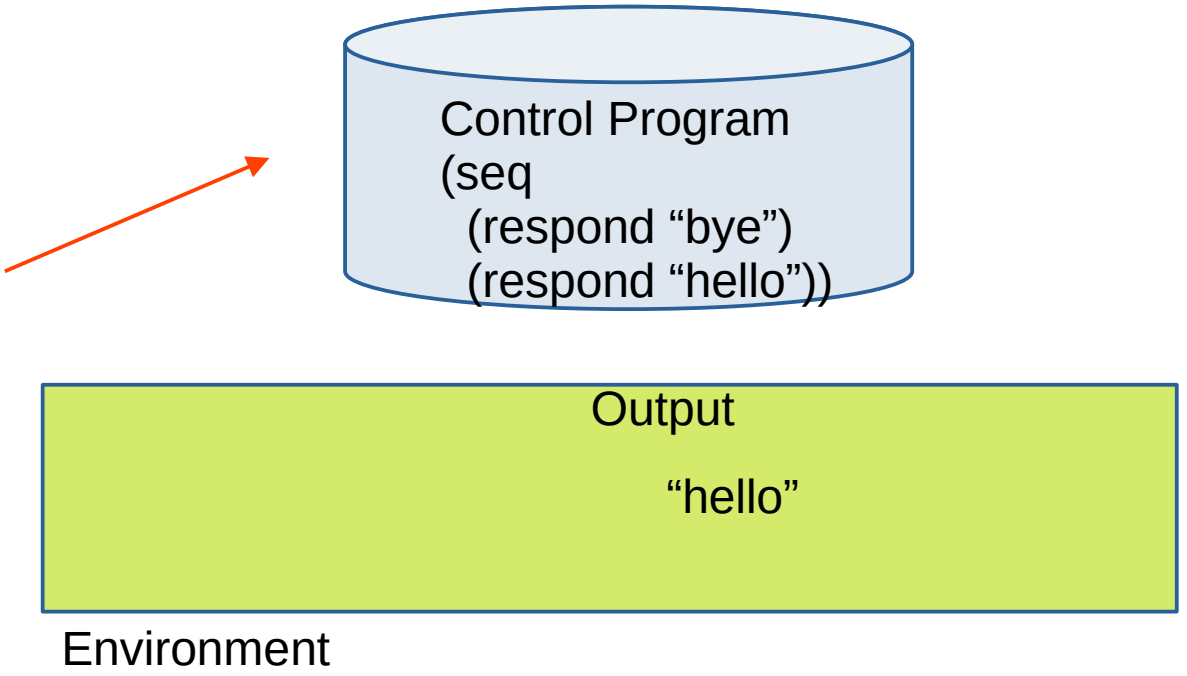
CP:
1. (respond “hello”)
2. (respond “bye”)
3. (seq CP CP)

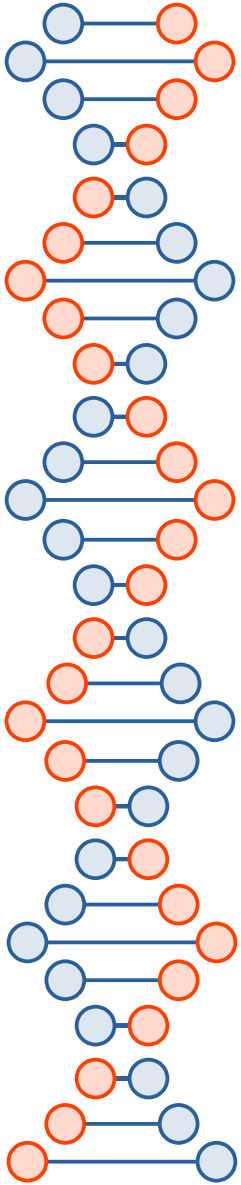
(seq (respond “hello”) (respond “hello”))

(seq (respond “bye”) (respond “hello”) (respond “bye”))

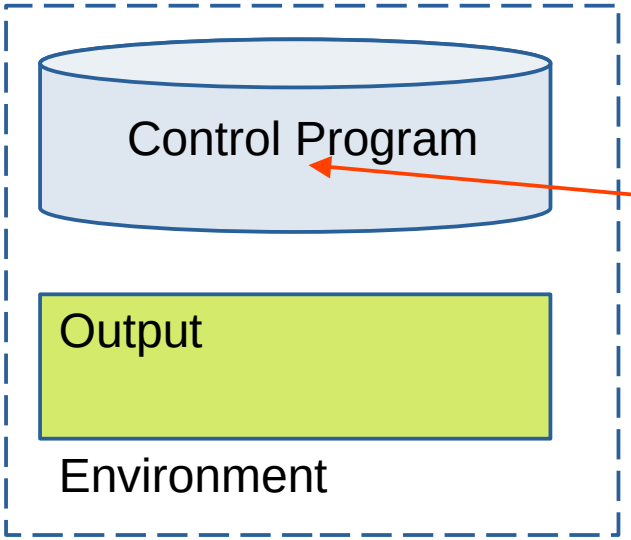


Part 1: Creating Models





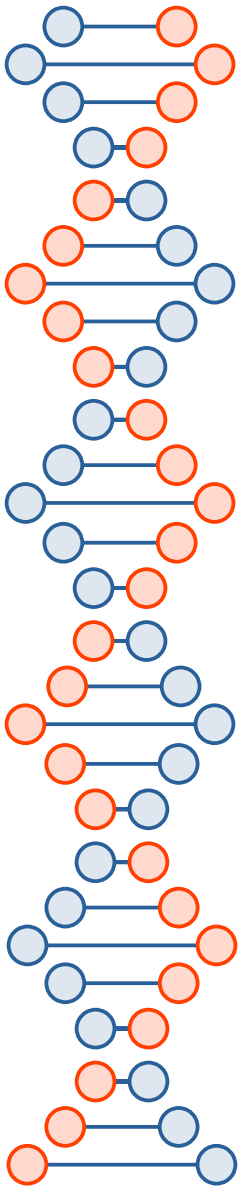
Part 1: Creating Models



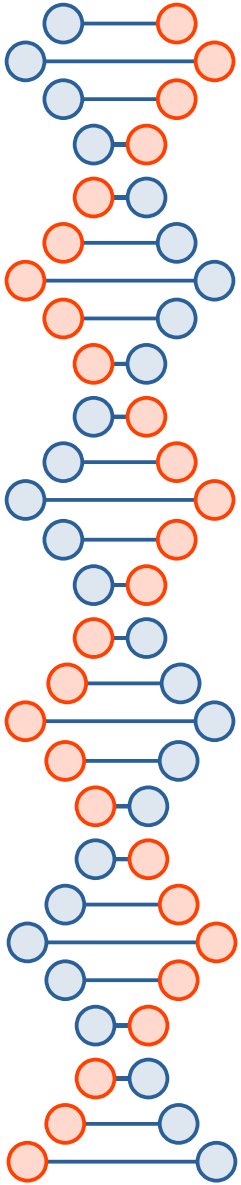
CP:
1. (respond “hello”)
2. (respond “bye”)
3. (seq CP CP)

There are an infinite number of possible models.

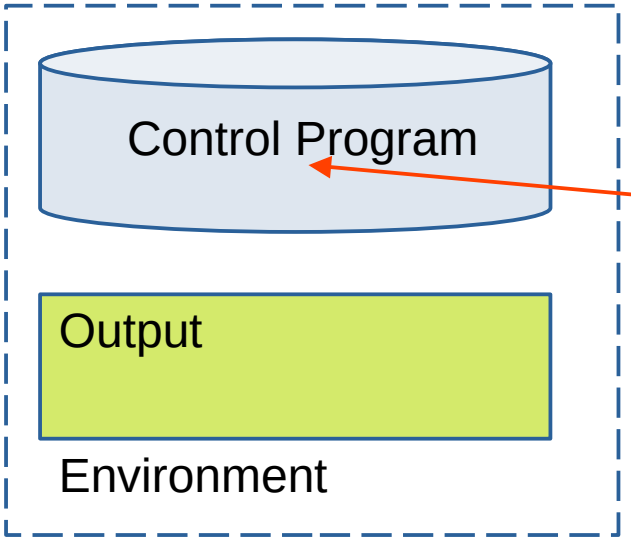
The correct models are those which respond “hello” last.



```
(defun interpret (operator md)
  (case (operator-label operator)
    (:respond-hello
      (setf (model-response md) "hello"))
    (:respond-bye
      (setf (model-response md) "bye"))
    (:seq
      (interpret (first (operator-children operator)) md)
      (interpret (second (operator-children operator)) md))
    (otherwise
      (error "interpret: unknown operator ~a" (operator-label operator)))))
```



Part 1: Creating Models

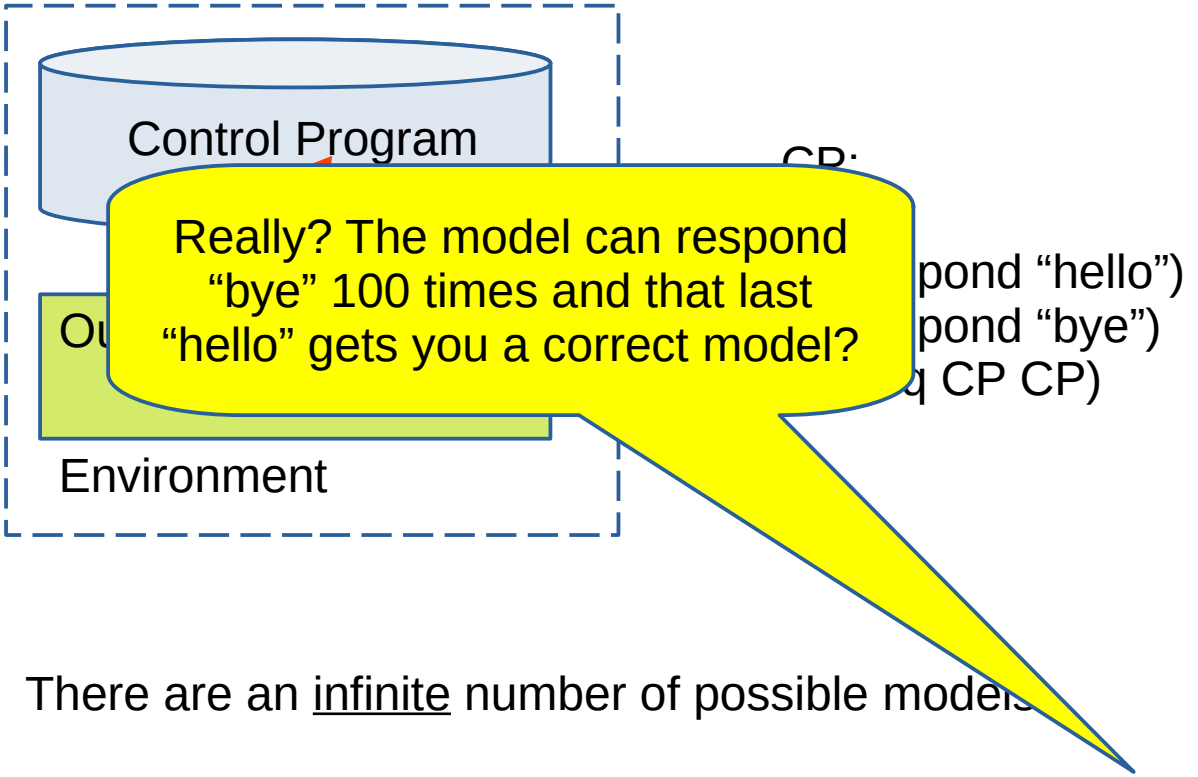


CP:
1.(respond “hello”)
3. (respond “bye”)
4. (seq CP CP)

There are an infinite number of possible models.

The correct models are those which respond “hello” last.

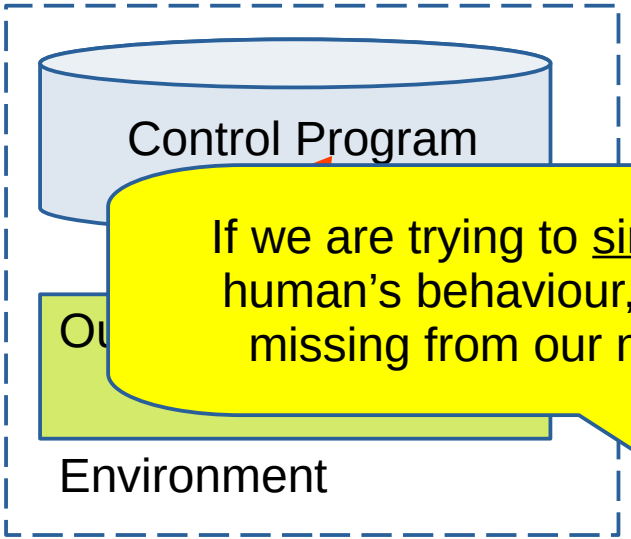
Part 1: Creating Models



There are an infinite number of possible models

The correct models are those which respond “hello” last.

Part 1: Creating Models

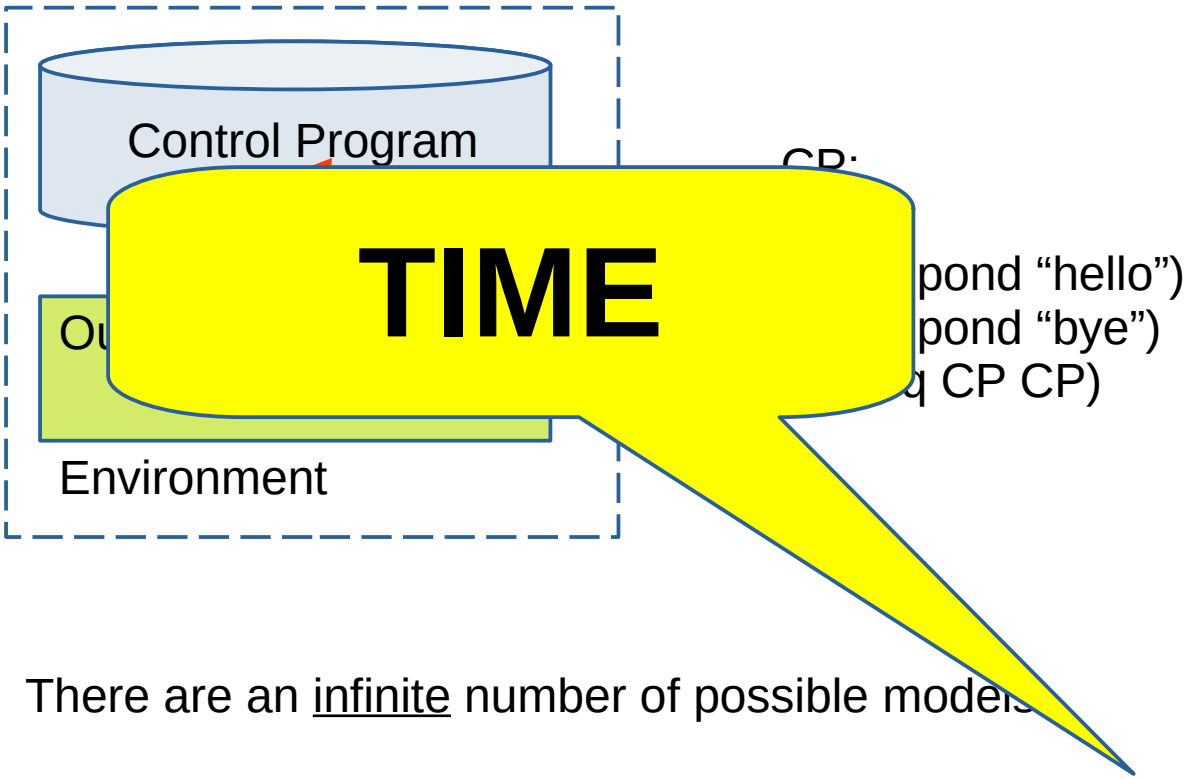


If we are trying to simulate a human's behaviour, what is missing from our model?

There are an infinite number of possible models

The correct models are those which respond "hello" last.

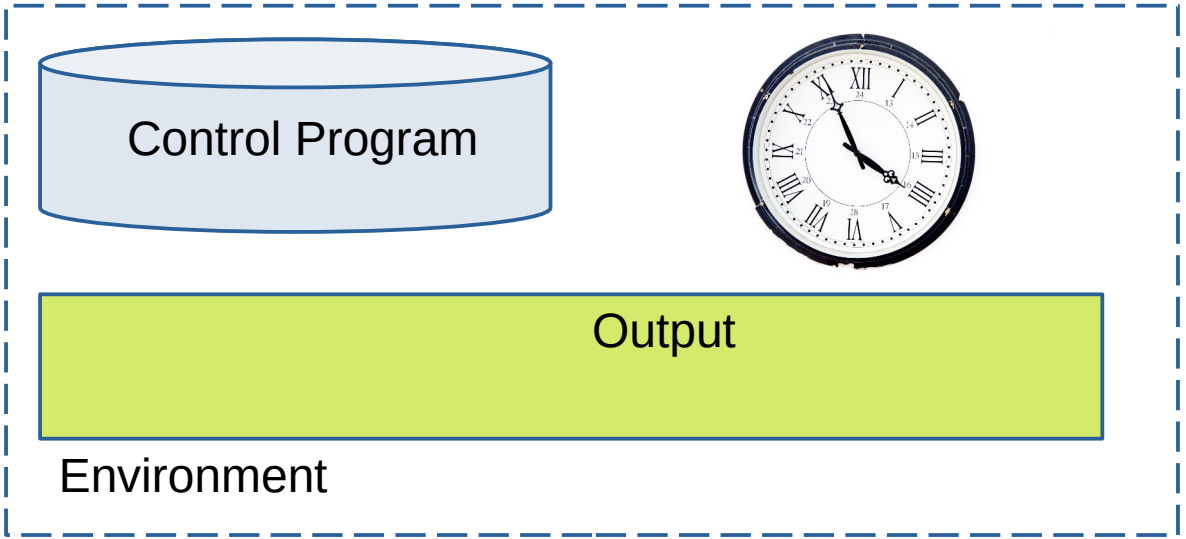
Part 1: Creating Models



There are an infinite number of possible models

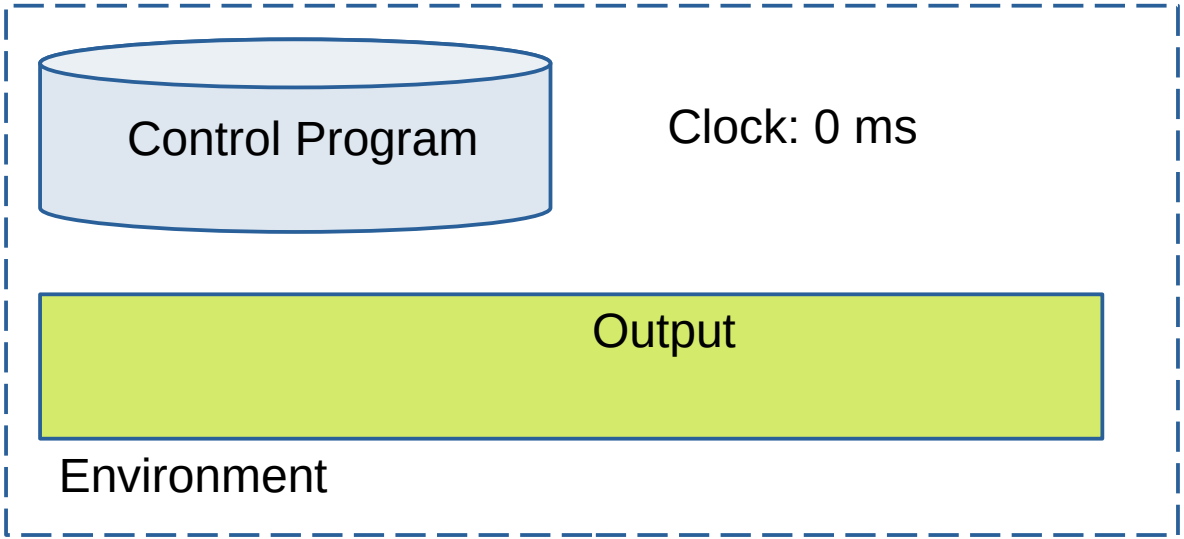
The correct models are those which respond “hello” last.

Part 1: Creating Models



- Clock starts at 0
- For each step in the control program, the clock is updated.
- Inputs/Outputs occur “at” a given time.
- Evaluation Function can use time when deciding correctness.

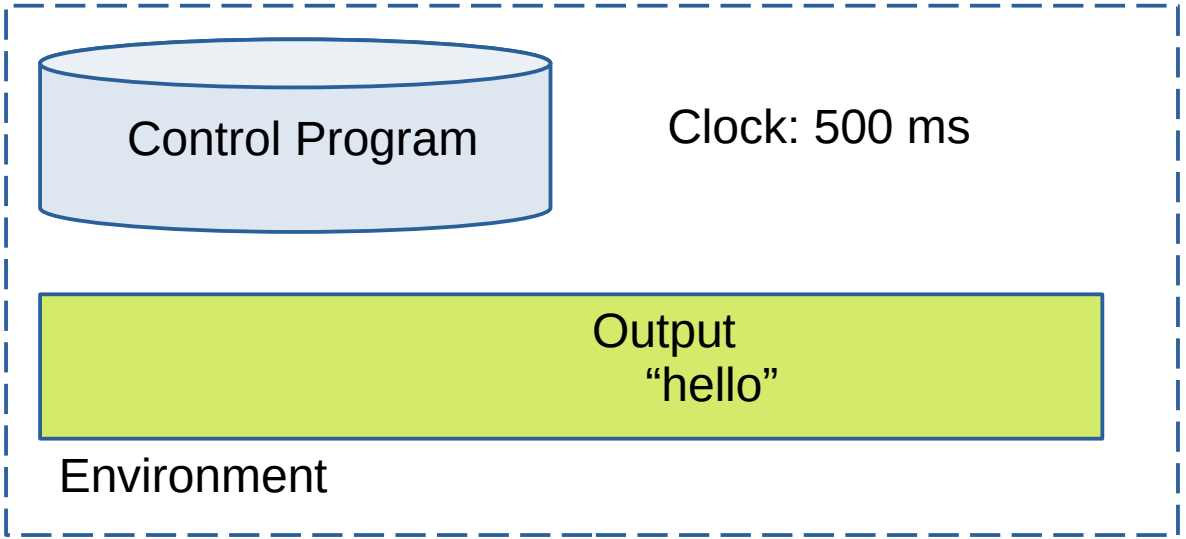
Part 1: Creating Models



Control Program:

(respond “hello”)

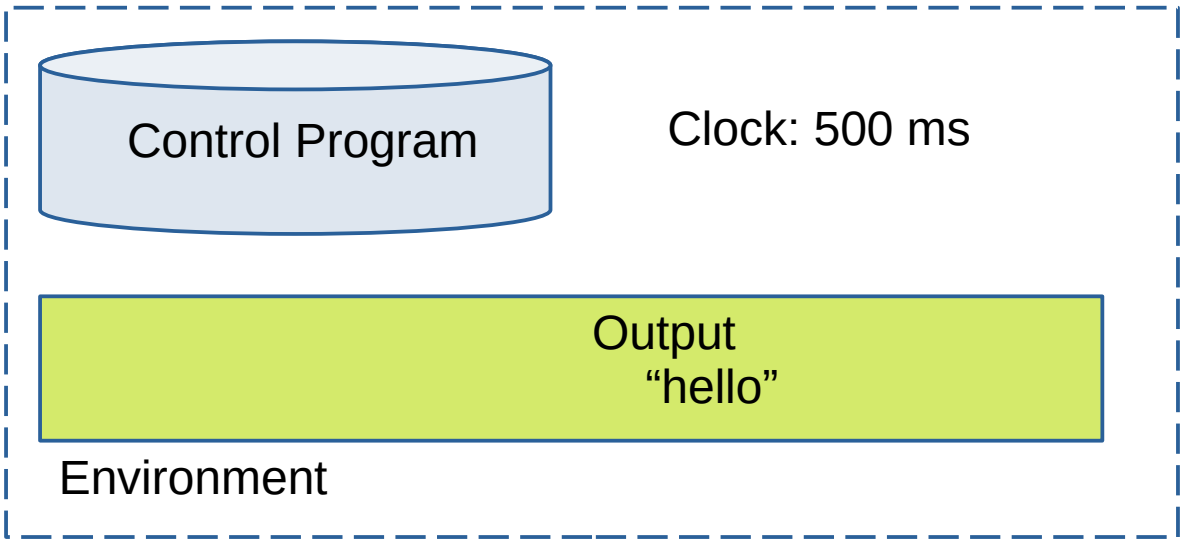
Part 1: Creating Models



Control Program:

(respond “hello”)

Part 1: Creating Models



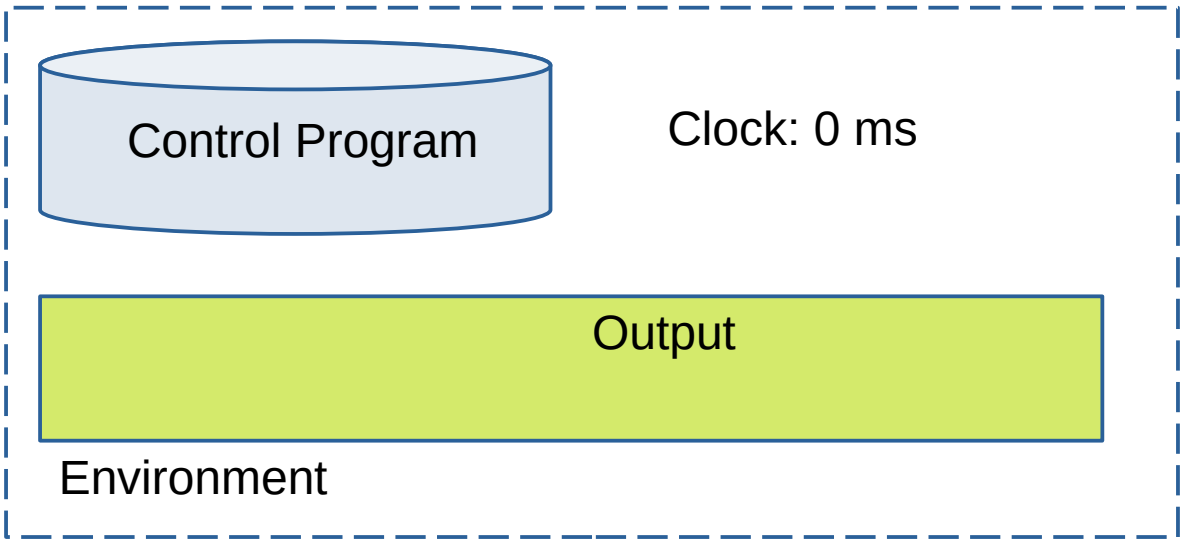
Control Program:

(respond “hello”)

Evaluation Function:

Is output “hello” within time 400-800ms?

Part 1: Creating Models



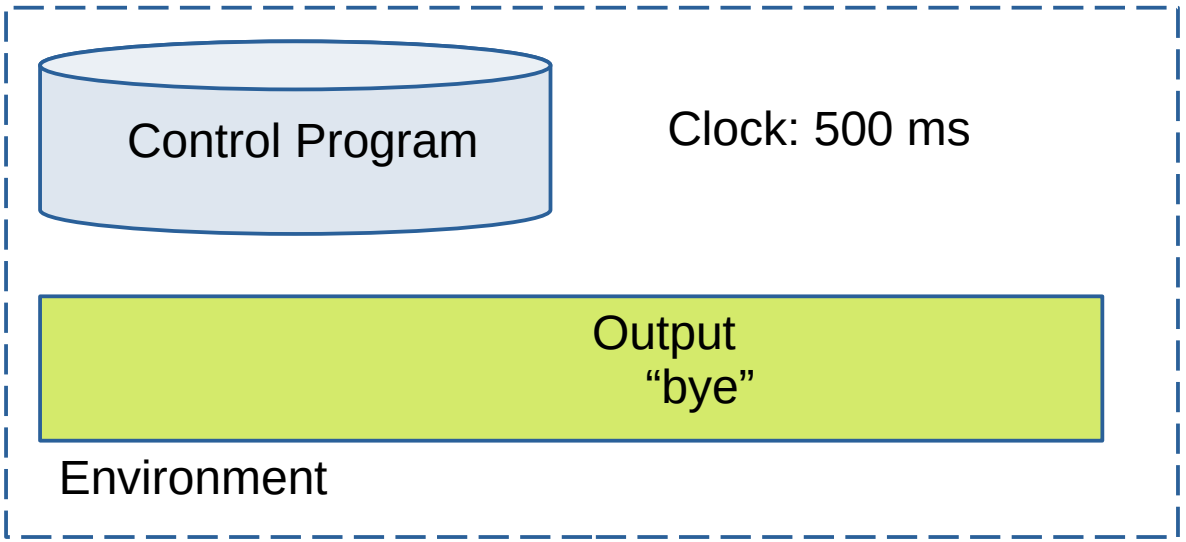
Control Program:

```
(seq (respond “bye”)
     (respond “hello”))
```

Evaluation Function:

Is output “hello” within time 400-800ms?

Part 1: Creating Models



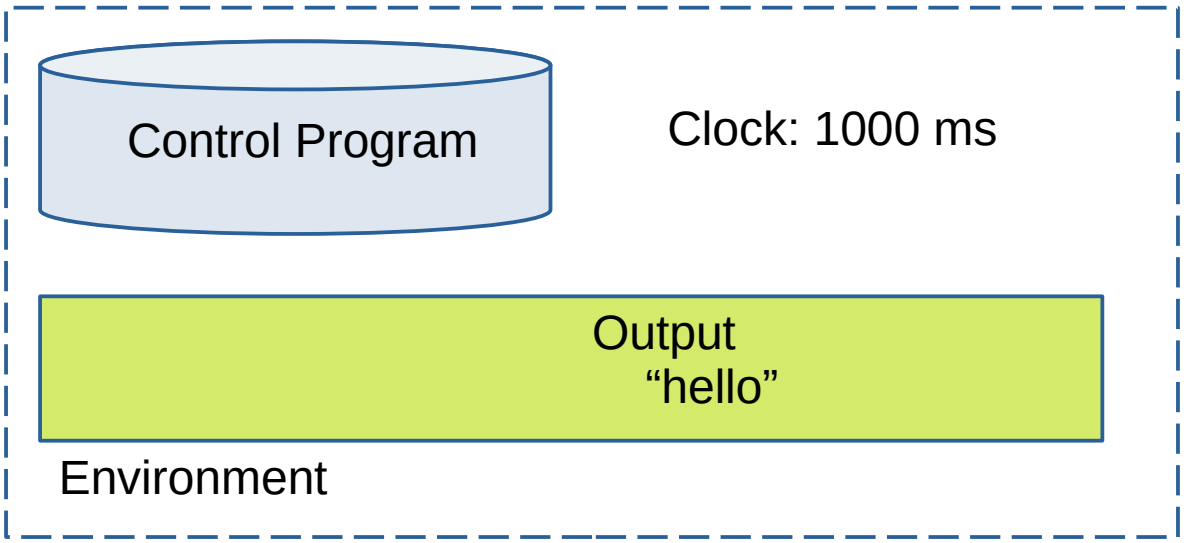
Control Program:

(seq (respond “bye”)
(respond “hello”))

Evaluation Function:

Is output “hello” within time 400-800ms?

Part 1: Creating Models

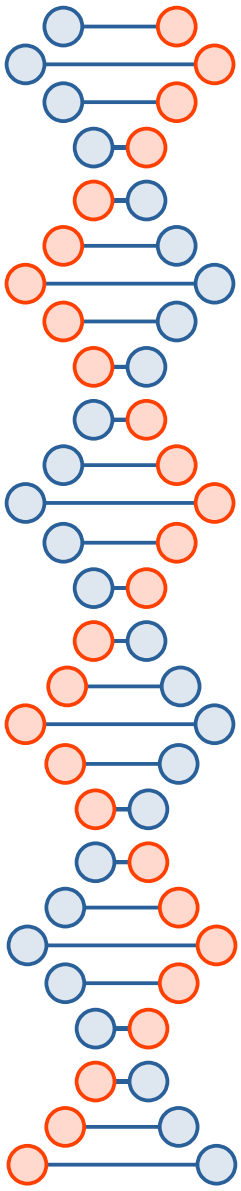


Control Program:

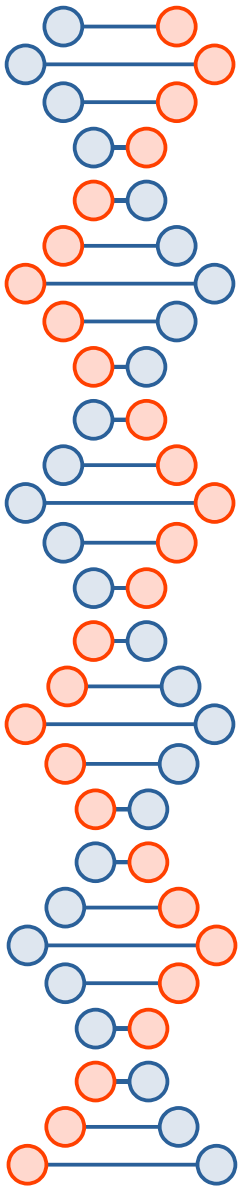
(seq (respond “bye”)
(respond “hello”))

Evaluation Function:

Is output “hello” within time 400-800ms?

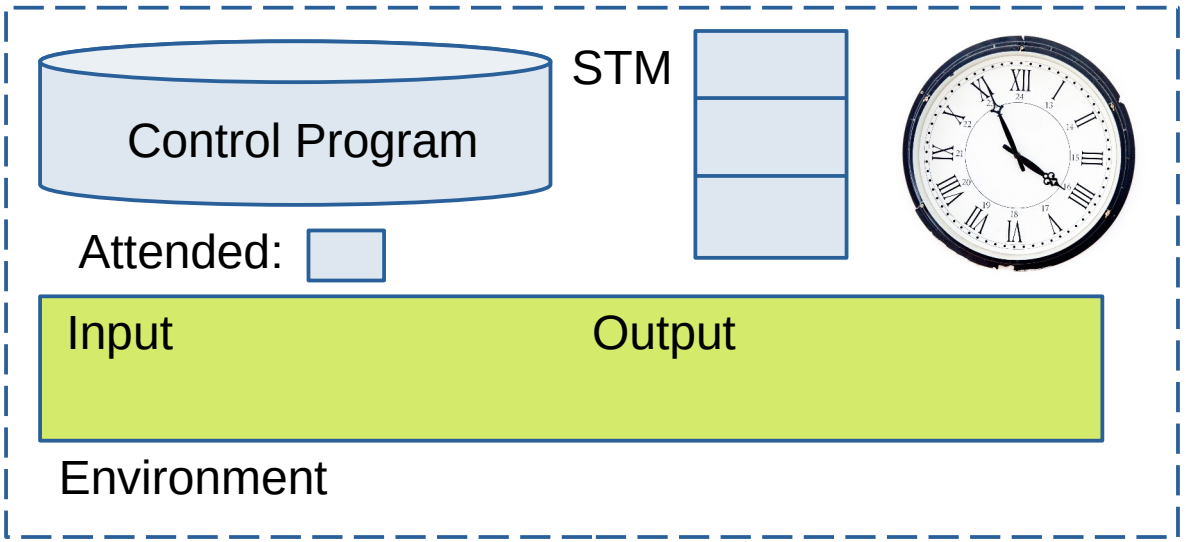


```
(defstruct model
  clock
  response ; hold response from model
  )

(defun evaluate-program (program)
  (let* ((result (run-experiment program))
         (result-response (first result))
         (result-time (second result)))
    (if (and (equalp "hello" result-response)
             (< result-time 800)
             (< 400 result-time))
        0 1) ; compute fitness
    ))

(defun run-experiment (program)
  (let ((md (make-model :clock 0 :response "")))
    (interpret program md)
    (list (model-response md)
          (model-clock md))))
```



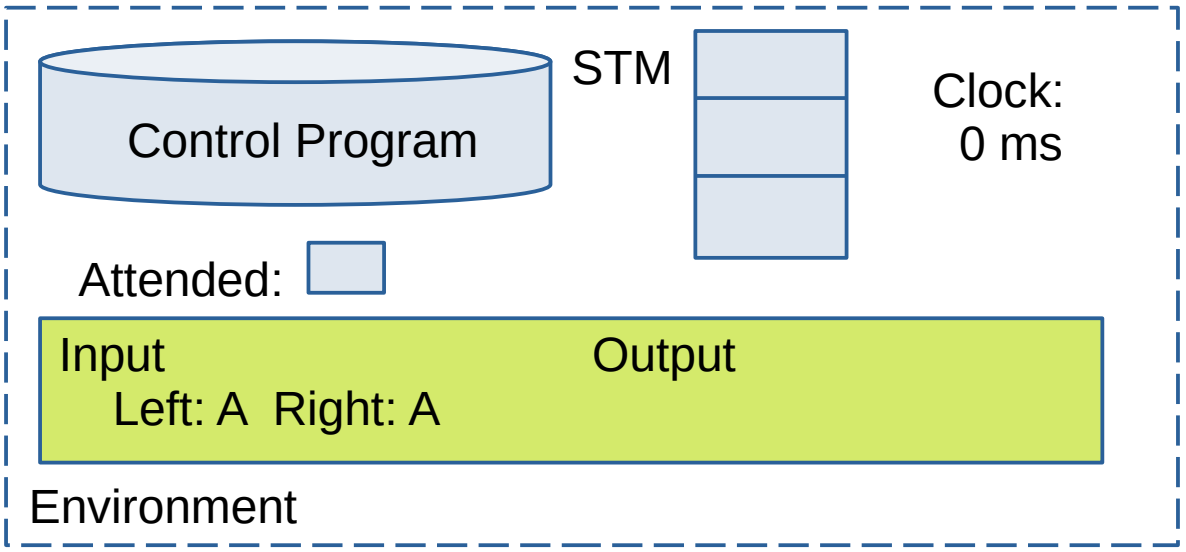
```
(defun interpret (operator md)
  (case (operator-label operator)
    (:respond-hello
      (incf (model-clock md) 500)
      (setf (model-response md) "hello"))
    (:respond-bye
      (incf (model-clock md) 500)
      (setf (model-response md) "bye"))
    (:seq
      (interpret (first (operator-children operator)) md)
      (interpret (second (operator-children operator)) md))
    (otherwise
      (error "interpret: unknown operator ~a" (operator-label operator)))))
```

Part 1: Creating Models



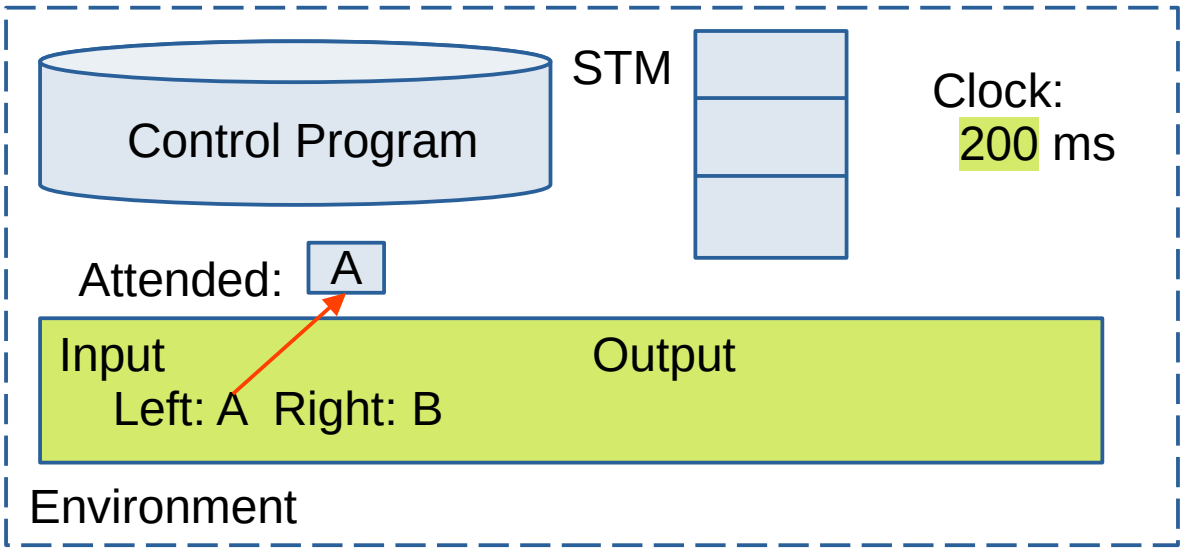
- Input: the model can retrieve input from the environment (input-left)
- Attended: is a slot where a piece of information can be held
- STM: is a push-down stack storing information for as long as needed

Part 1: Creating Models



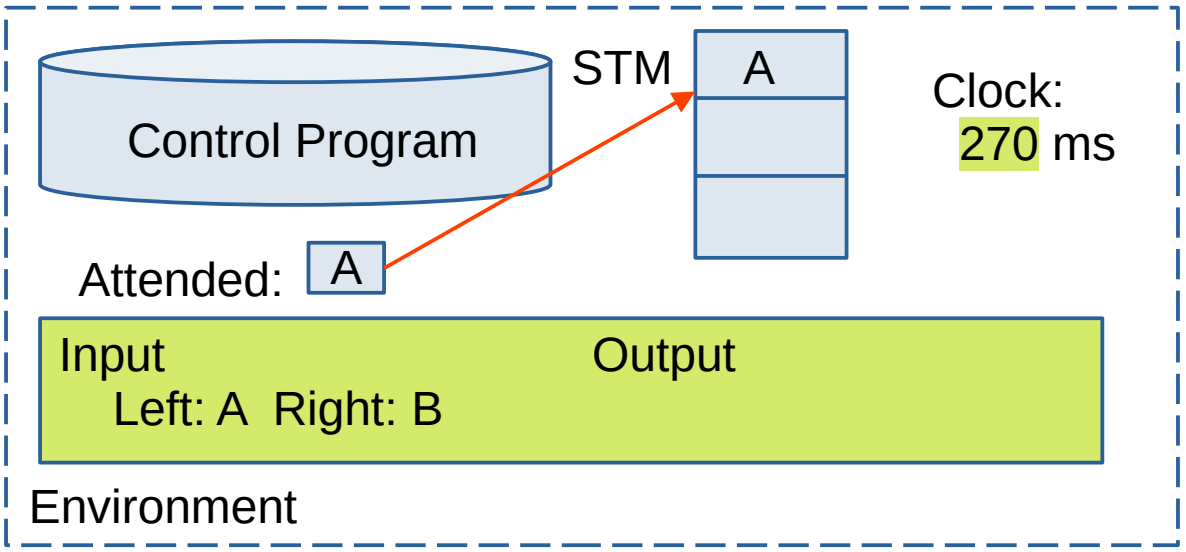

```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models



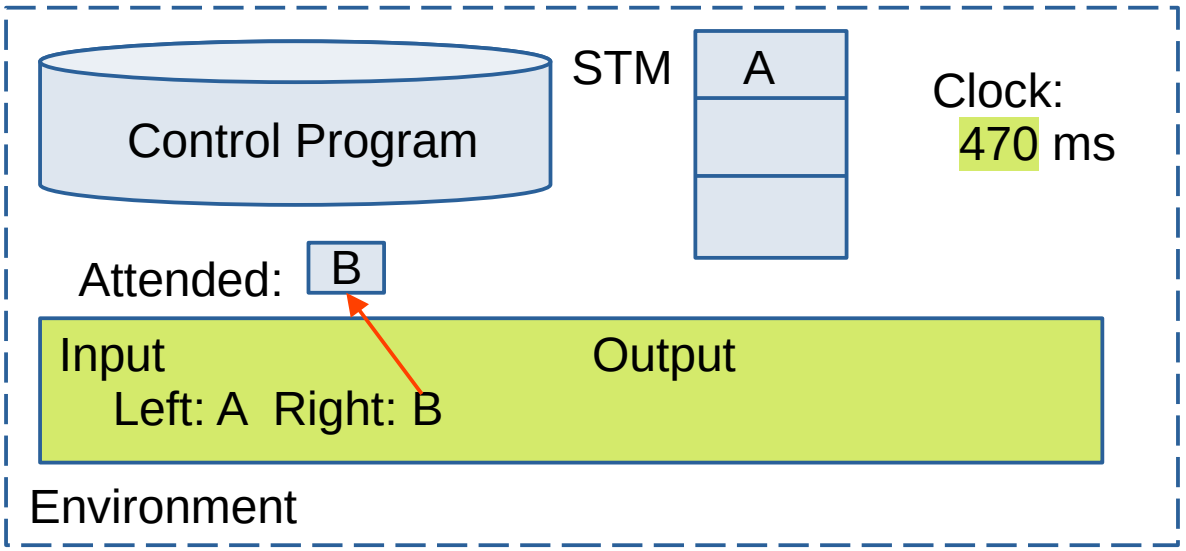

```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models



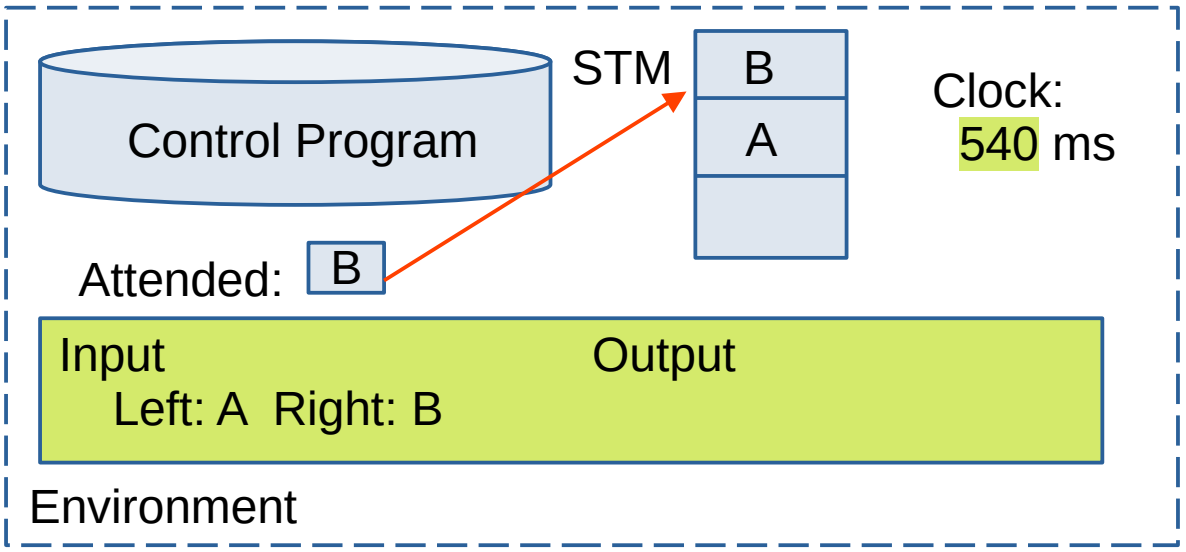

```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models



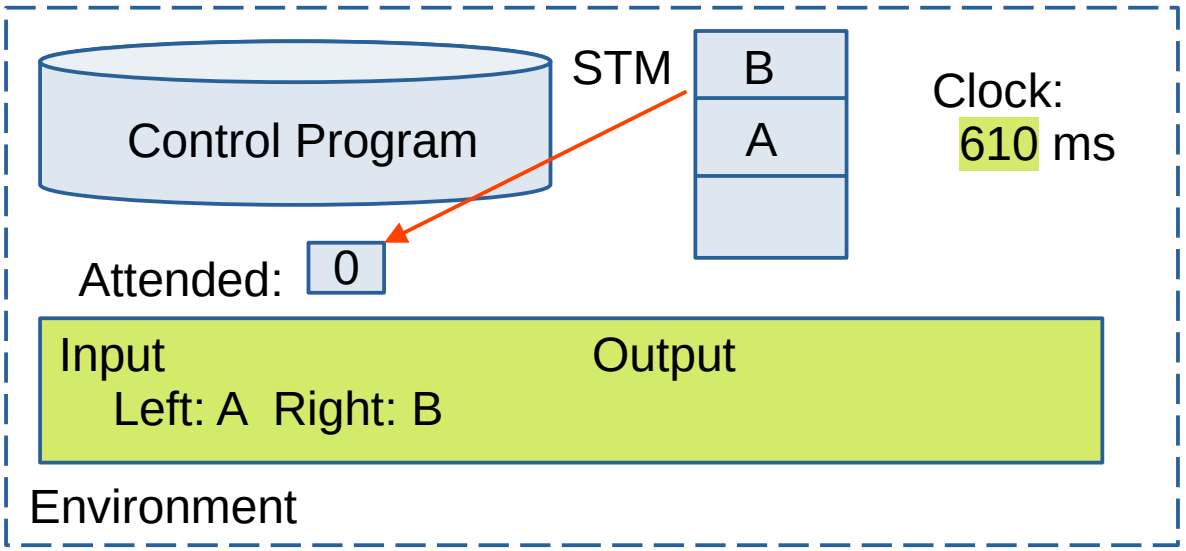

```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models



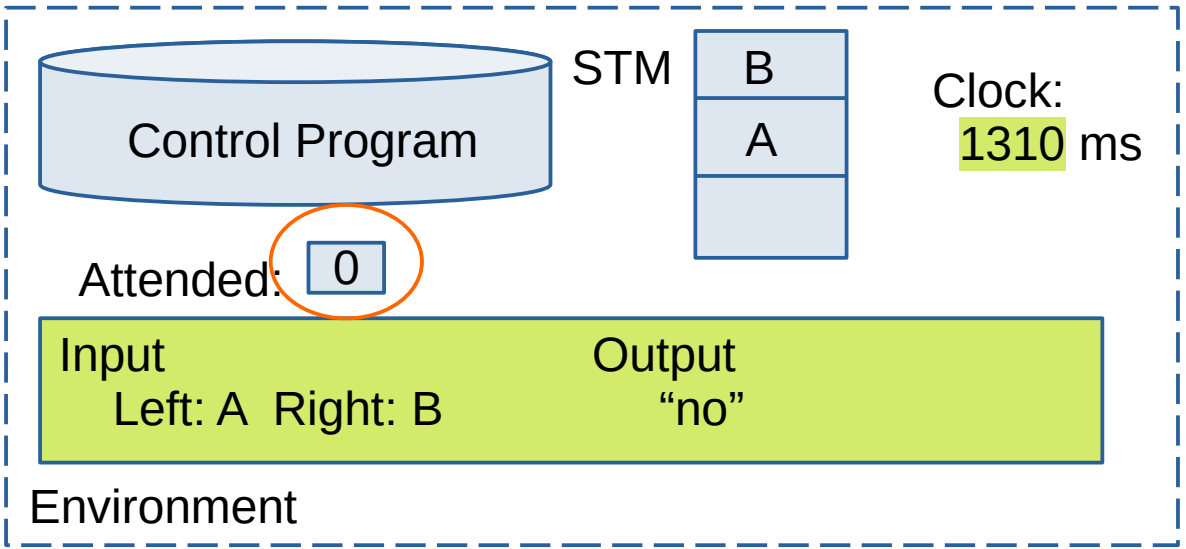

```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models




```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Part 1: Creating Models




```
(seq (input-left)
  (seq (put-stm)
    (seq (input-right)
      (seq (put-stm)
        (seq (cmp-1-2)
          (if (respond “yes”)
              (respond “no”))))))
```

Delayed Match to Sample (1)

Delayed Match to Sample (2)

Delayed Match to Sample (3)

Time

Delayed Match to Sample (4)

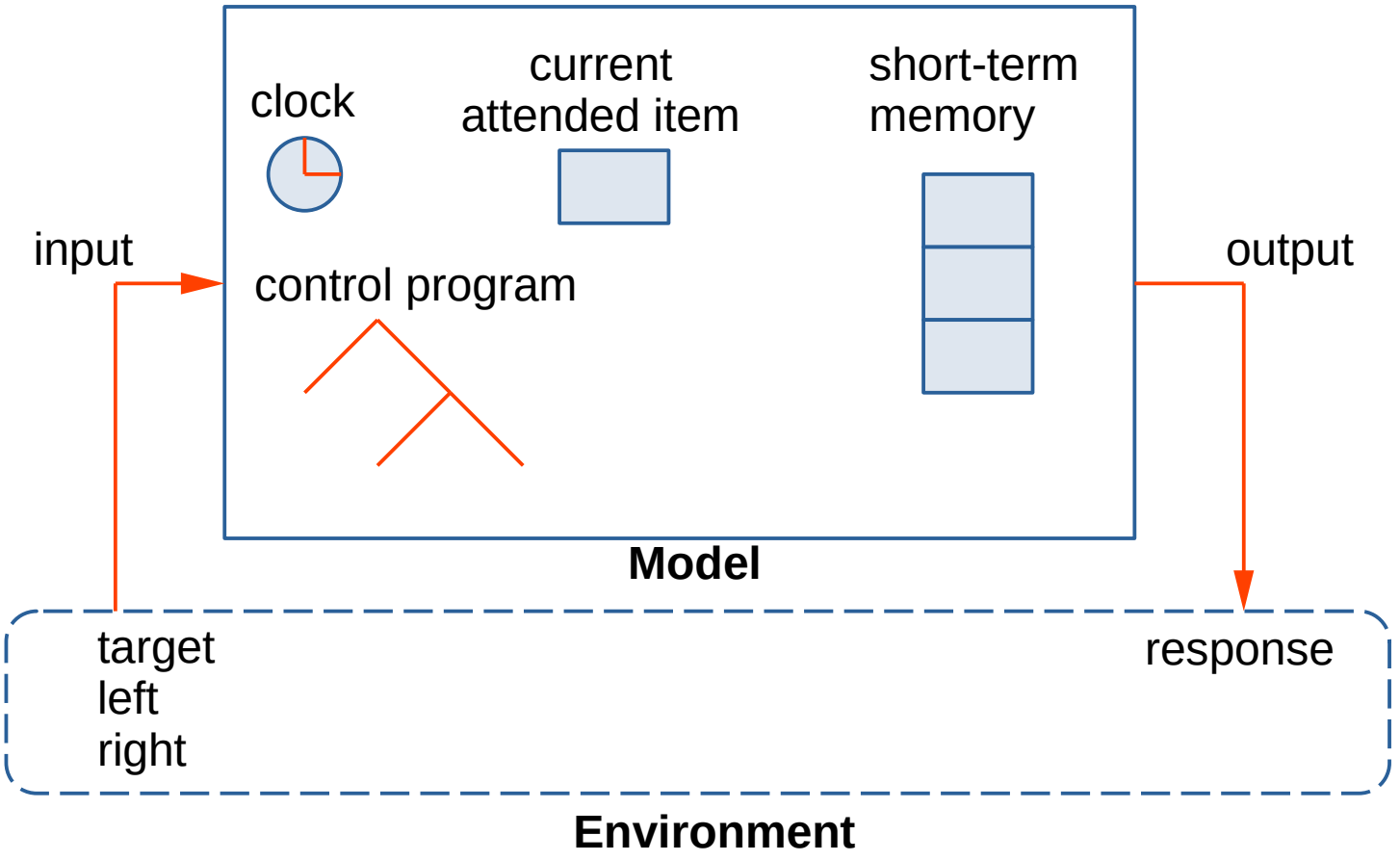
Delay

Time

Accuracy: 95.7%
Response time: 767ms

(Chao et al, 1999)

Structure of Model



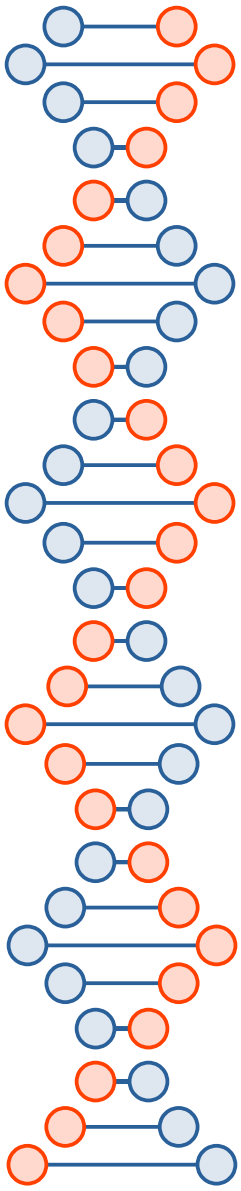
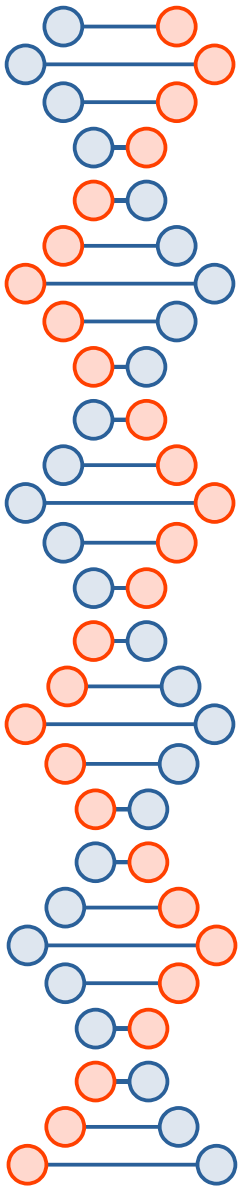


Table 1: Overview of operators used in DMTS models.

Name	Function	Type
input-X	sets model 'current' to value of left/right/target input, if it is visible	input
respond-X	sets model 'response' to "R"/"L", if inputs are visible	output
access-N	sets model 'current' to STM item N ($N \in \{1,2,3\}$)	stm
compare-M-N	compares value of STM items M and N ($M \neq N \in \{1,2,3\}$) and sets 'current' to 1 if equal, or 0 if not	cognitive
nil	sets model 'current' to 0	cognitive
put-stm	pushes value in model 'current' to top of STM	stm
dotimes-N	repeats a given expression ($N \in \{2,3,5\}$)	syntax
if	executes condition, executes one of two expressions based on value in model 'current'	syntax
prog-N	sequence of expressions ($N \in \{2,3,4\}$)	syntax
wait-N	advances model clock ($N \in \{25,50,100,200,1000,1500\}$)	syntax

```
(prog-4 (wait-500) (input-target) (put-stm)
      (prog-4   (wait-1000)
                (input-left)
                (put-stm)
                (if    (compare-1-2)
                       (respond-L)
                       (respond-R)))))
```



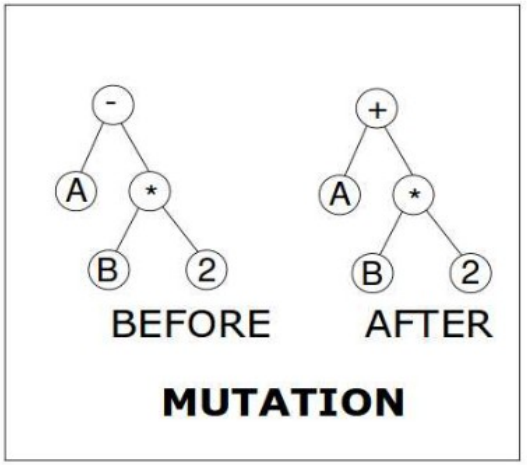
```
;; Defines the state of the model
(defstruct model
  clock current stm ; base model
  inputs response ; I/O requirements for DMTS task
  )

;; Collect results from running model (defined by operator (program) + md).
(defun interpret (operator md)
  (unless (> (model-clock md) 10000) ; time-out - adjust this if required
    (case (operator-label operator)

      (:input-left
        (incf (model-clock md) (timings-input (model-timings md)))
        (when (> (model-clock md) start-input)
          (setf (model-current md) (second (model-inputs md)))))

      (:input-right
        (incf (model-clock md) (timings-input (model-timings md)))
        (when (> (model-clock md) start-input)
          (setf (model-current md) (third (model-inputs md)))))

      (:input-target
        (incf (model-clock md) (timings-input (model-timings md)))
        (when (<= (model-clock md) end-target)
          (setf (model-current md) (first (model-inputs md)))))

      (:respond-left
```

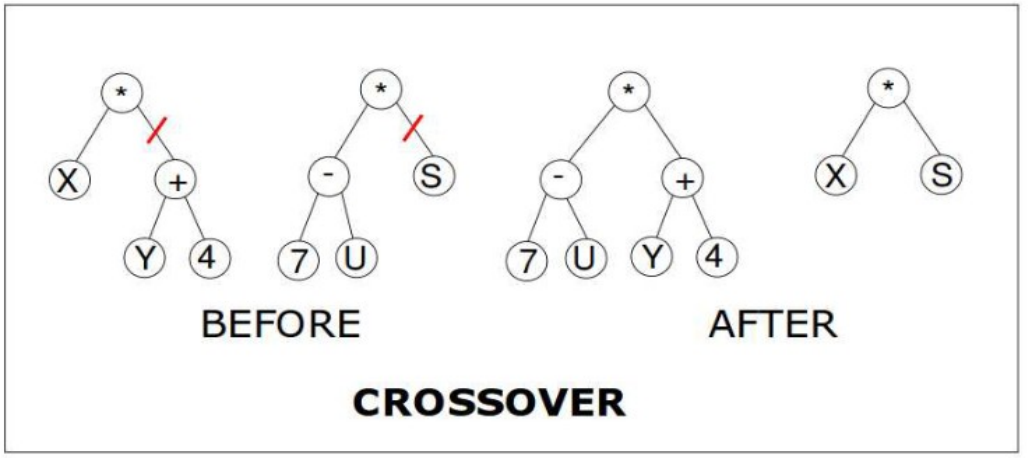
```
;; Holds information about results of experiment
(defstruct result
  inputs response accuracy timing)
```

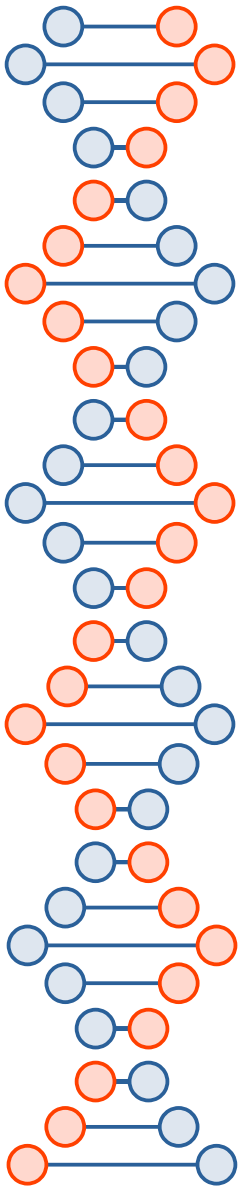
```
;; Run a single experiment against the given program, returning information on performance.
(defun run-experiment (program)
  (let ((results '())
        (expt-data (alexandria:shuffle *data*)))
    (dolist (input expt-data)
      (let ((md (make-model :clock 0 :current 0 :stm '(0 0 0)
                            :inputs input :response "-")))
        (interpret program md)
        (let ((result (make-result :inputs input :response "-" :accuracy 0 :timing 0)))
          (when (> (model-clock md) start-input) ; when clock is after allowed time for response
            (setf (result-response result) (model-response md)) ; record model's response
            (setf (result-accuracy result) ; record whether it is correct or not
                  (if (string= (result-response result) (target-response input))
                    1
                    0))
            (setf (result-timing result) (- (model-clock md) start-input)) ; record the response time
            )
          (push result results))))
    results))
```

Part 2: Program Synthesis - Genetic Programming

1. Creates a population of models
2. Evaluates all the models using a *fitness function*
3. Selects the best models, and generates a new population by altering/combining existing models.
4. Repeats until termination condition reached.



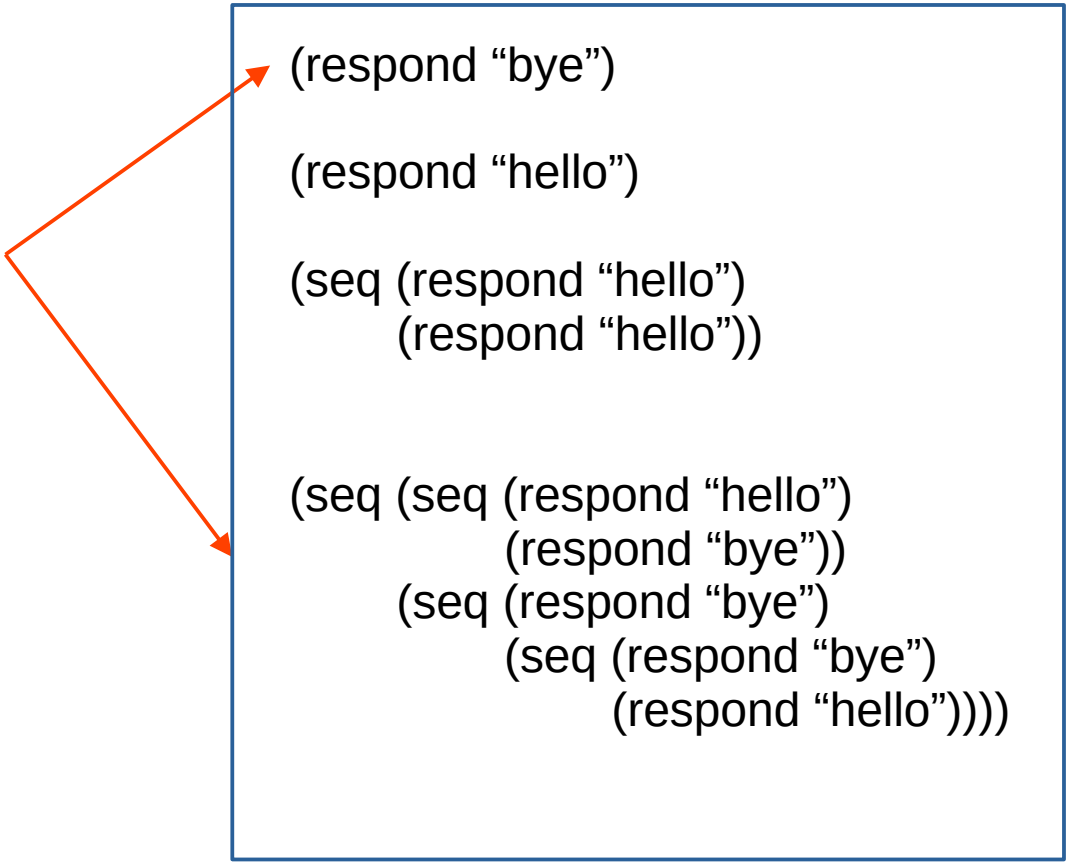




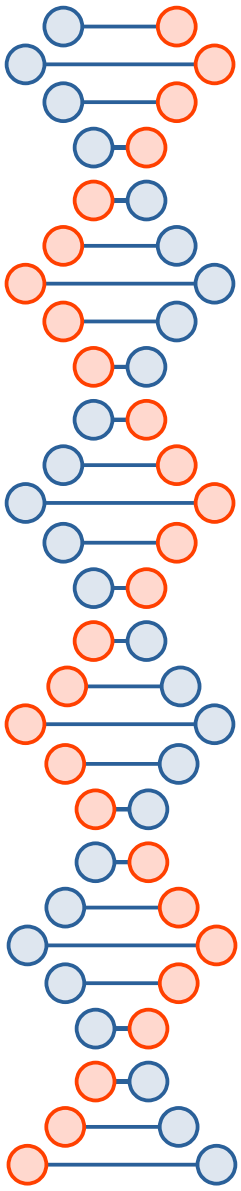
Part 2: Program Synthesis

CP grammar:

1. (respond “hello”)
2. (respond “bye”)
3. (seq CP CP)



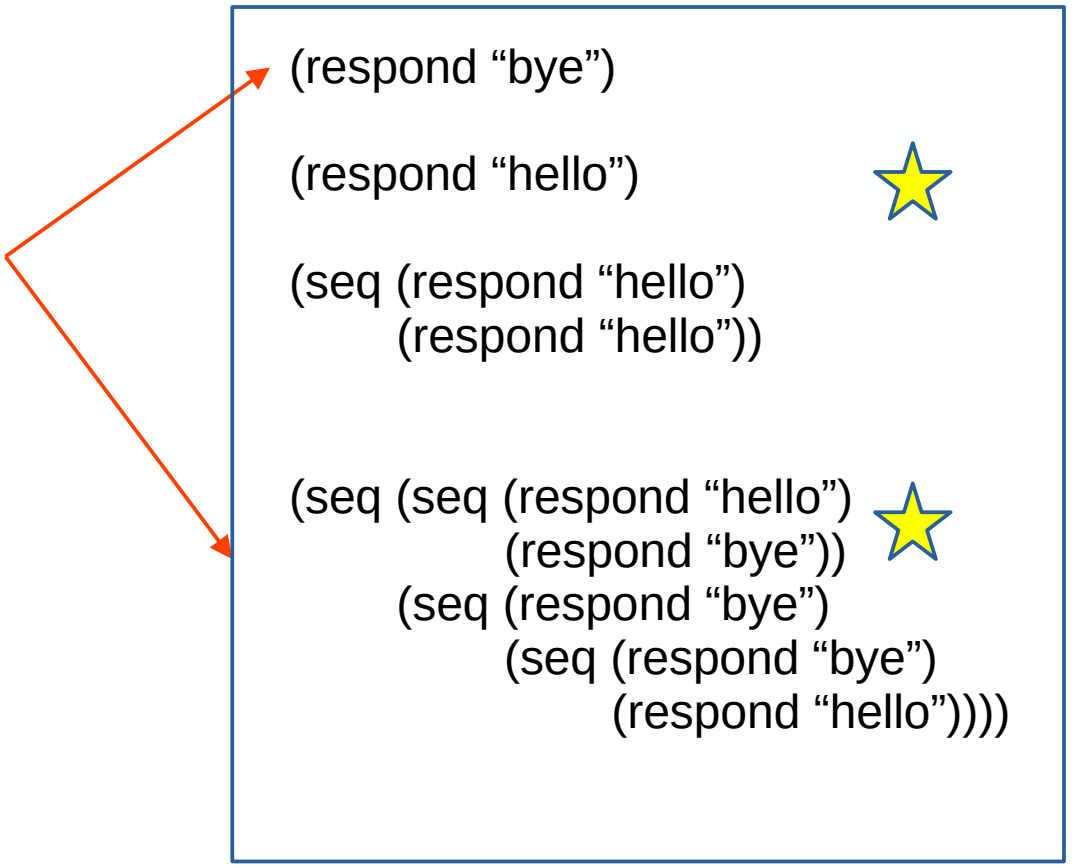
Random Population



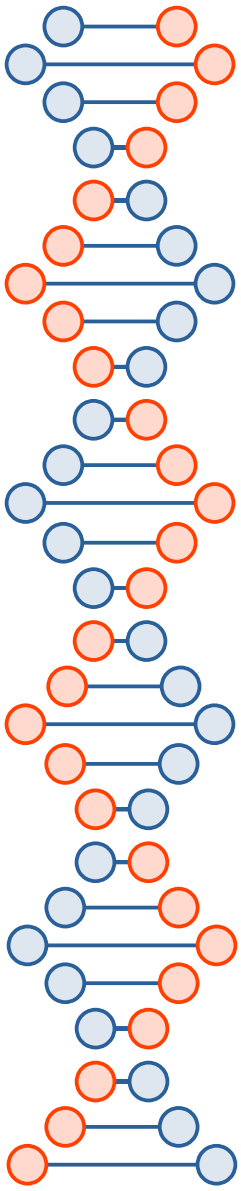
Part 2: Program Synthesis

CP grammar:

1. (respond “hello”)
2. (respond “bye”)
3. (seq CP CP)



Random Population



Part 2: Program Synthesis

(respond “bye”)

(respond “hello”)

(seq (respond “hello”)
(respond “hello”))

(seq (seq (respond “hello”)
(respond “bye”))
(seq (respond “bye”)
(seq (respond “bye”)
(respond “hello”))))

Random Population

(respond “bye”)

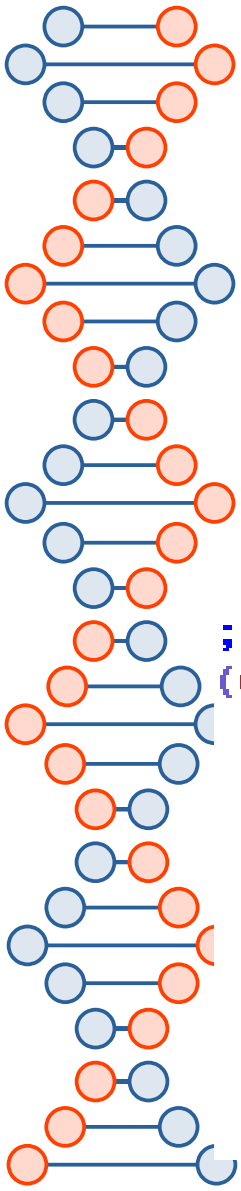
(seq (seq (respond “hello”)
(respond “bye”))
(respond “hello”))

(respond “hello”)

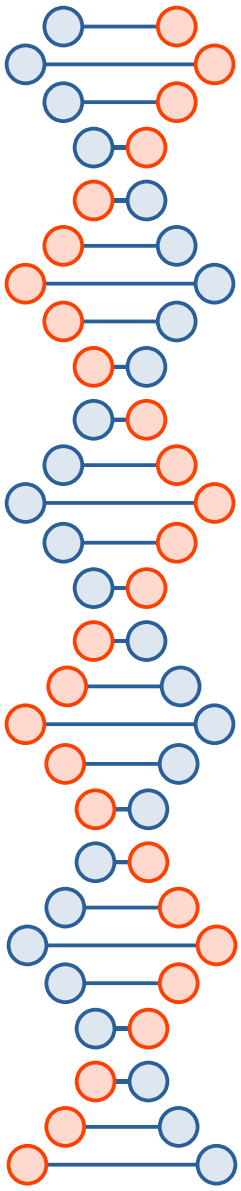
(seq (respond “bye”)
(respond “bye”))

(seq (respond “hello”)
(respond “bye”))

New Population



```
;; Returns a dotted list holding the available operators and number of childrer
(defun operator-set ()
  '((INPUT-LEFT . 0)
    (INPUT-RIGHT . 0)
    (INPUT-TARGET . 0)
    (RESPOND-LEFT . 0)
    (RESPOND-RIGHT . 0)
    (ACCESS-STM-1 . 0)
    (ACCESS-STM-2 . 0)
    (ACCESS-STM-3 . 0)
    (COMPARE-1-2 . 0)
    (COMPARE-2-3 . 0)
    (COMPARE-1-3 . 0)
;; Runs the GP system with given parameters, results logged to files.
(defun run-gp (&key (logger nil)) ; logger function
  (setf *phase* 1) ; initial phase for phased-evolution
  (gems:launch (operator-set) #'evaluate-program
               :total-generations *total-generations*
               :population-size *population-size*
               :initial-depth 1
               :maximum-depth 10
               :elitism t
               :type :steady-state
               :logger logger))
```



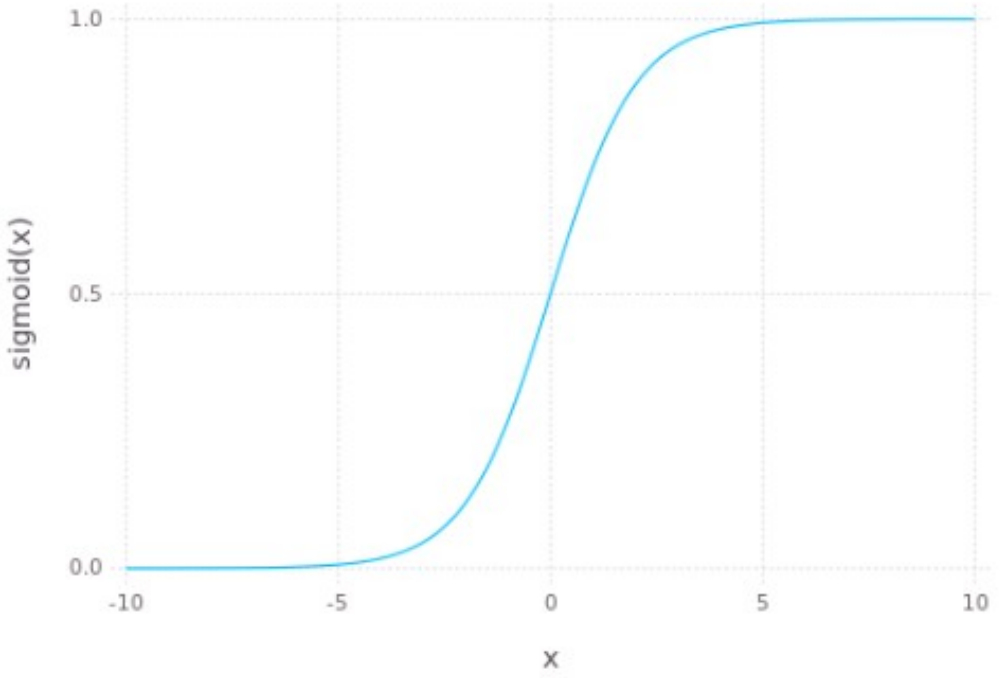
Multi-Objective Fitness Function

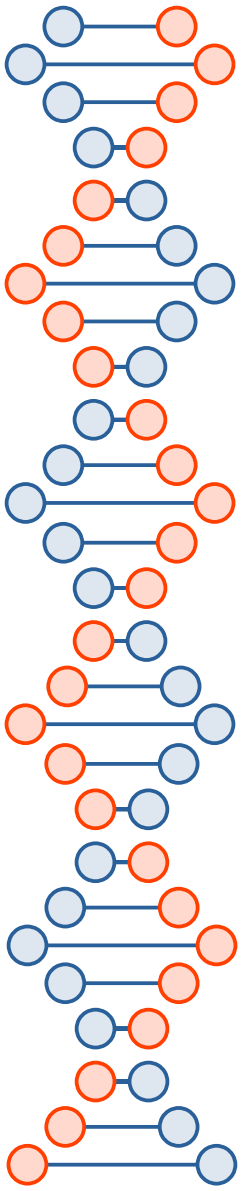
- accuracy to match that of humans - 95.7%
- response time to match that of humans - 767ms
- program size, to favour smaller models
- overall fitness must be in range [0.0, 1.0], with 0.0 representing the best model

- As accuracy is a number in range [0-1.0], we can represent this by:
  - f_a = |accuracy - 0.957| / 0.957

Fitness Function for Time

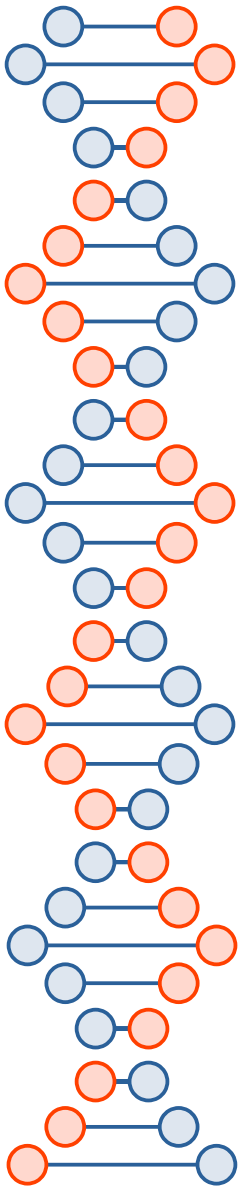
- Time can be any duration, hence we use a “squashing function”
-





Fitness Function for Time

- Time can be any duration, hence we use a “squashing function”
- f_t = half-sigmoid(|response-time - 767| / RT)
- RT controls how much the response-time will differ from the target before f_t gets close to 1.0



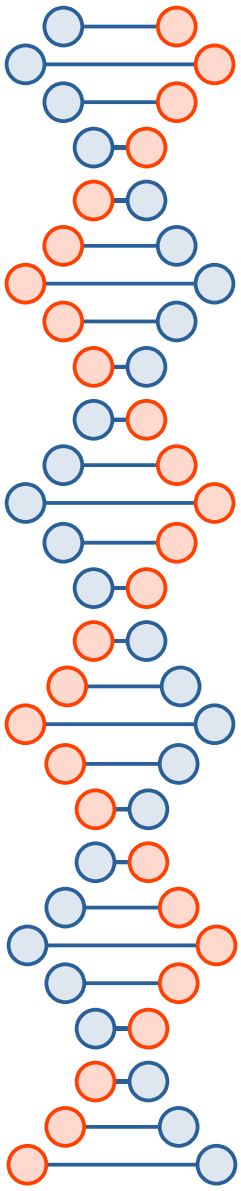
Phased Evolution (1)

1. $f_a = |\text{accuracy} - 0.957|/0.957$: this is the difference of the model's and target accuracy, scaled to the range $[0, 1]$.

2. $f_t = \text{half-sigmoid}(|\text{response-time} - 767|/\text{RT})$: this is the difference of the model's and target response time, with a variable scale factor RT.

3. $f_s = \text{half-sigmoid}(|\text{program-size} - 10|/\text{PS})$: this is the difference of the model's and an arbitrary target program size of 10, with a variable scale factor PS.

$$f = a \times f_a + b \times f_t + c \times f_s$$

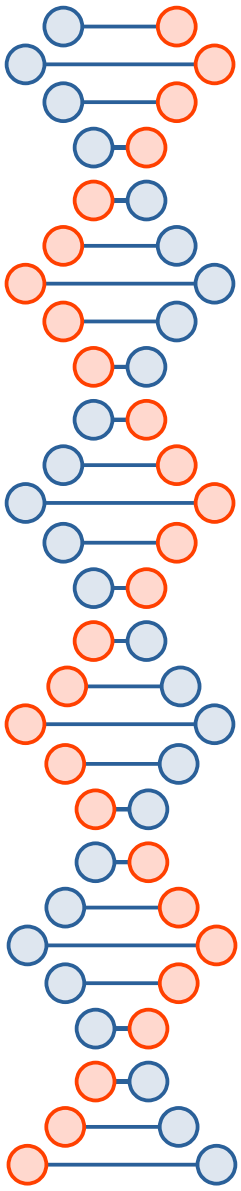


Phased Evolution (2)

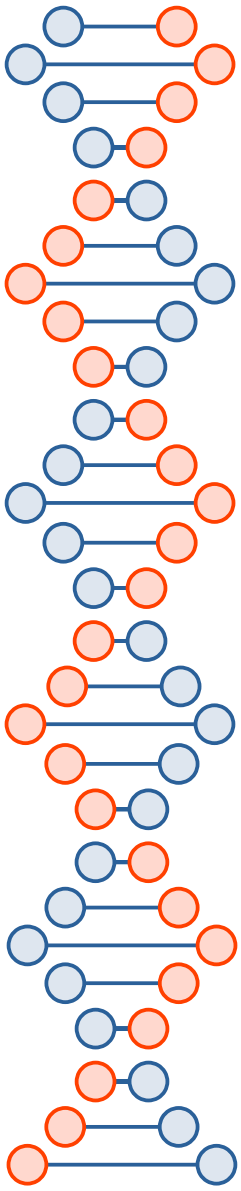
Phase 1 fitness $f = f_a$

Phase 2 fitness $f = (a \times f_a + b \times f_t)$, where $a + b = 1$

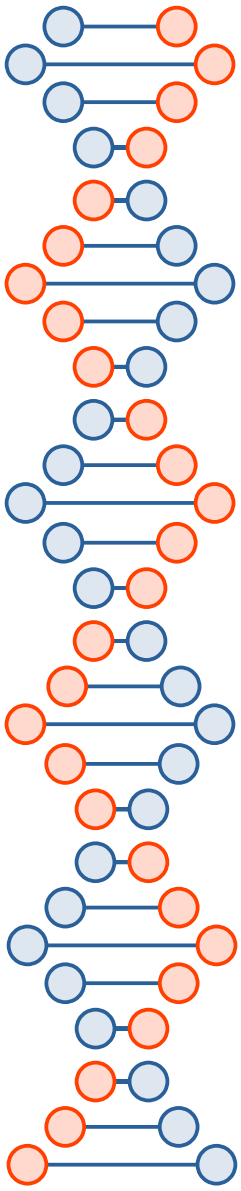
Phase 3 fitness $f = (a \times f_a + b \times f_t + c \times f_s)$, where $a + b + c = 1$



```
;; runs experiment on a single program
(defun evaluate-program (program)
  (let* ((results (run-experiment program))
         (accuracy (alexandria:mean (mapcar #'result-accuracy results)))
         (f-a (fitness-accuracy accuracy))
         (response-time (alexandria:mean (mapcar #'result-timing results)))
         (f-t (fitness-time response-time))
         (program-size (gems:program-size program))
         (f-s (fitness-size program-size)))
    (values  ; overall-fitness, optional extra information
      (overall-phased-fitness f-a f-t f-s)
      (list accuracy f-a response-time f-t program-size f-s *phase*) ; extra information
      )))
```

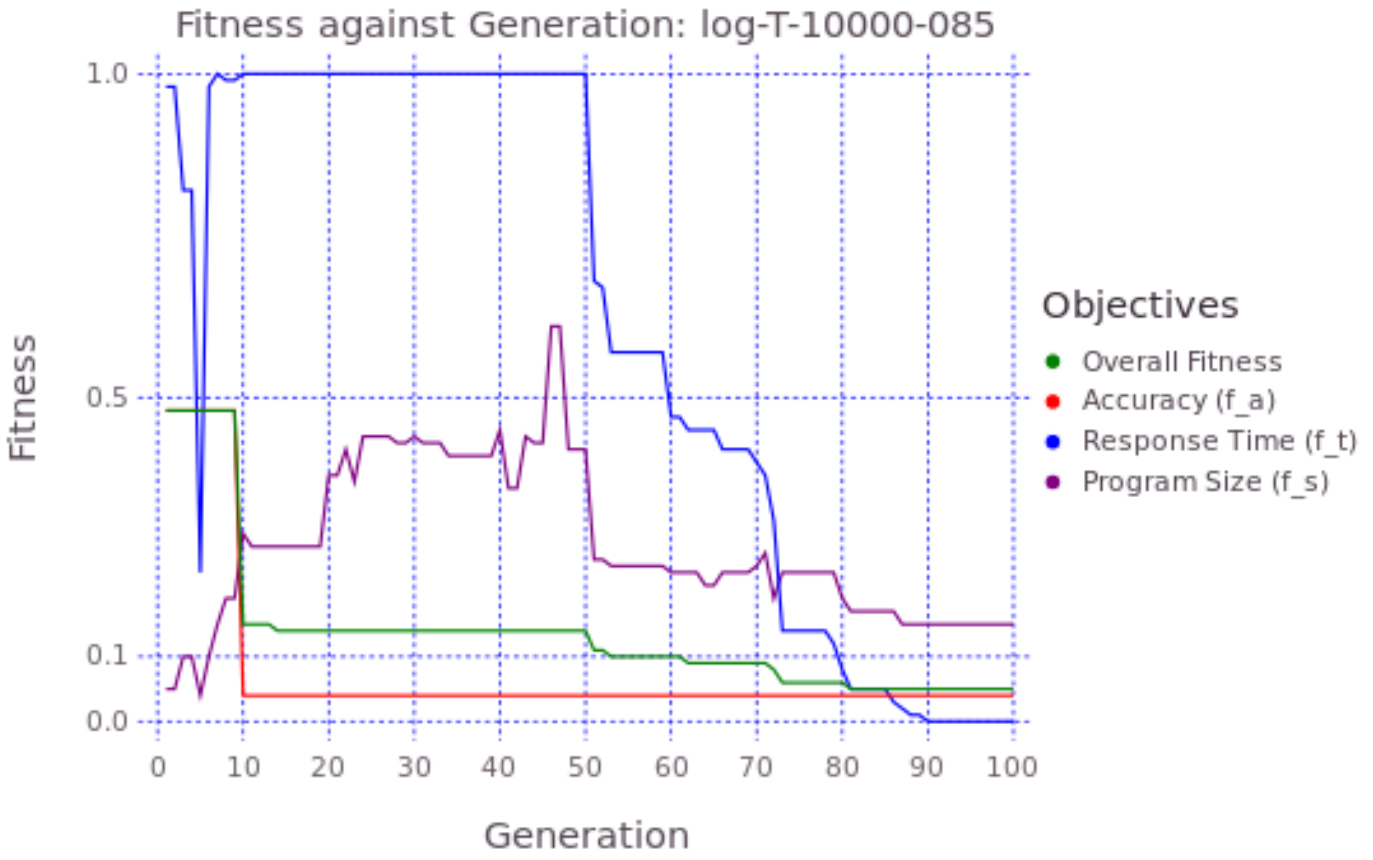


```
;; Computes the f_a objective function: 95.7% is target mean accuracy
(defun fitness-accuracy (performance)
  (/ (abs (- 0.957 performance))
     0.957))

;; Computes the f_t objective function: 767ms is target mean response
(defun fitness-time (response-time)
  (gems:half-sigmoid (/ (abs (- response-time 767))
                        *time-rt*)))

;; Computes the f_s objective function.
(defun fitness-size (program-size)
  (gems:half-sigmoid (/ program-size *size-ps*)))
```

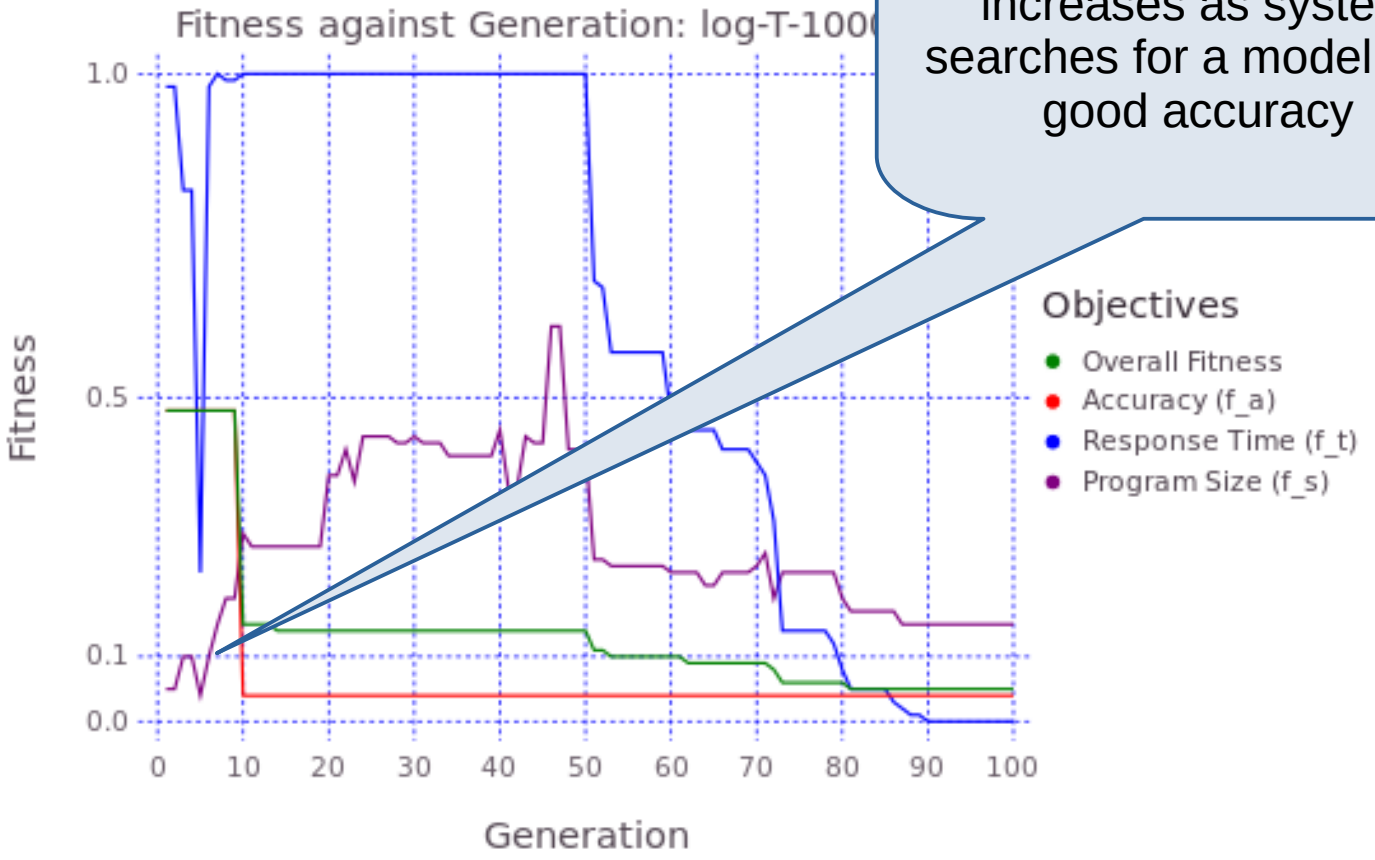


```
;; Computes the fitness for current phase
(defun fitness-for-phase (f-a f-t f-s)
  (case *phase*
    (1 ; single objective
     f-a)
    (2 ; two objectives
     (/ (+ (* *propn-fitness-accuracy* f-a)
           (* *propn-fitness-time* f-t))
        (+ *propn-fitness-accuracy* *propn-fitness-time*)))
    (otherwise ; all three objectives
      (+ (* *propn-fitness-accuracy* f-a)
         (* *propn-fitness-time* f-t)
         (* *propn-fitness-size* f-s)))))

;; Computes fitness, using the phases
(defun overall-phased-fitness (f-a f-t f-s)
  (when (and (< *phase* 3)
             (< (fitness-for-phase f-a f-t f-s) *good-model-threshold*))
    (incf *phase*))
  (fitness-for-phase f-a f-t f-s))
```

Phased Evolution (3)

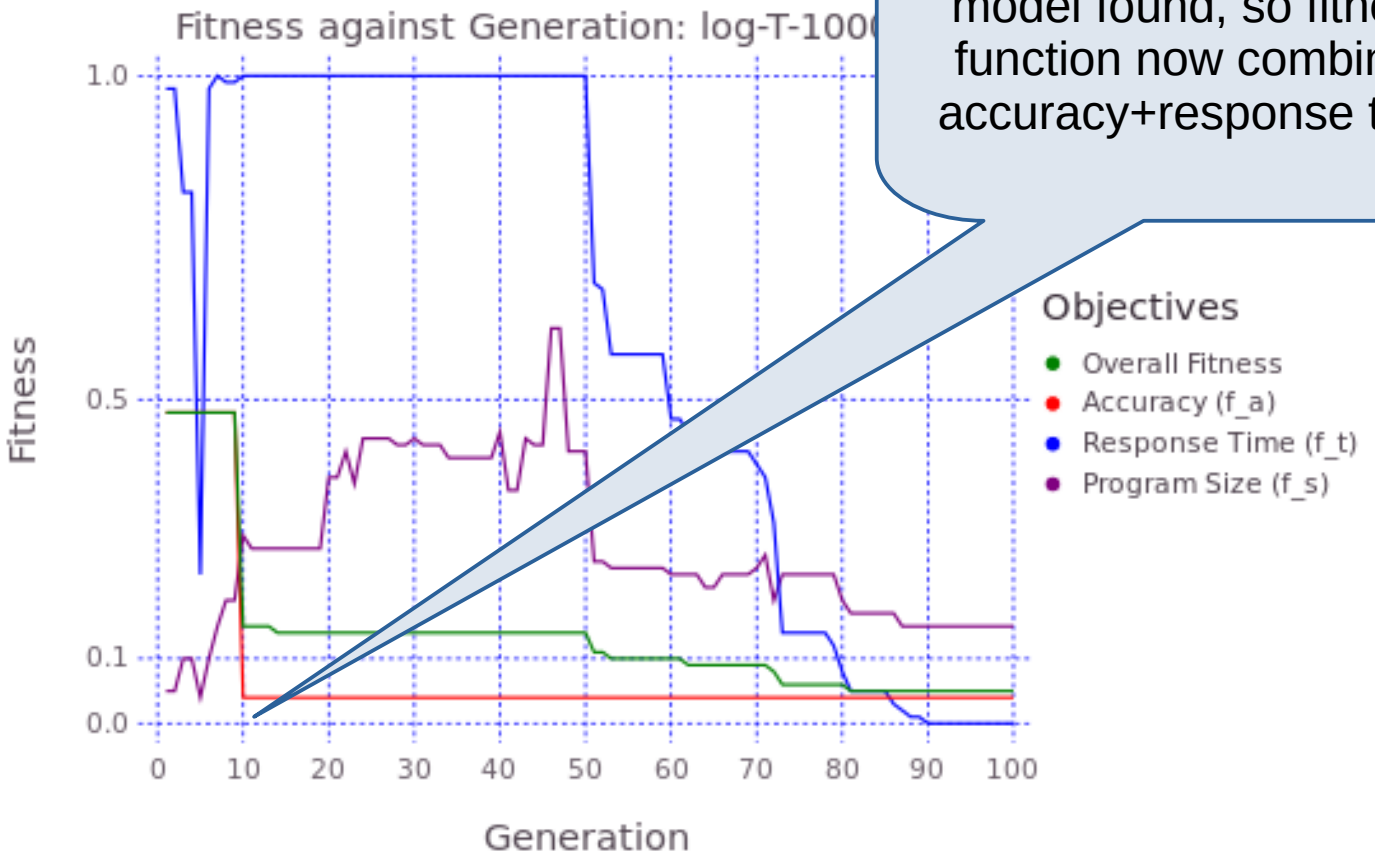


Phased Evolutio



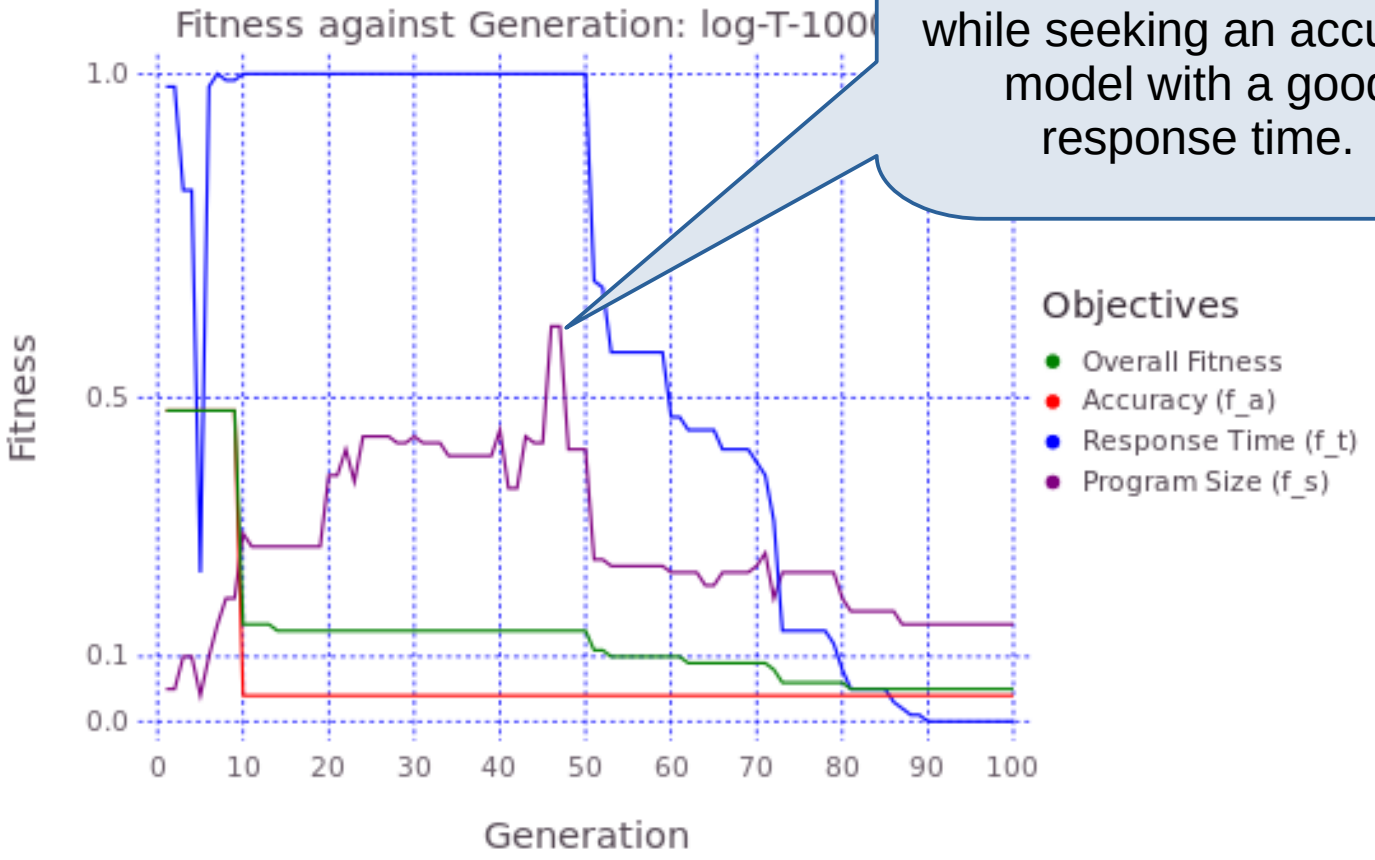
Phase 1: Program size increases as system searches for a model with good accuracy

Phased Evolutio



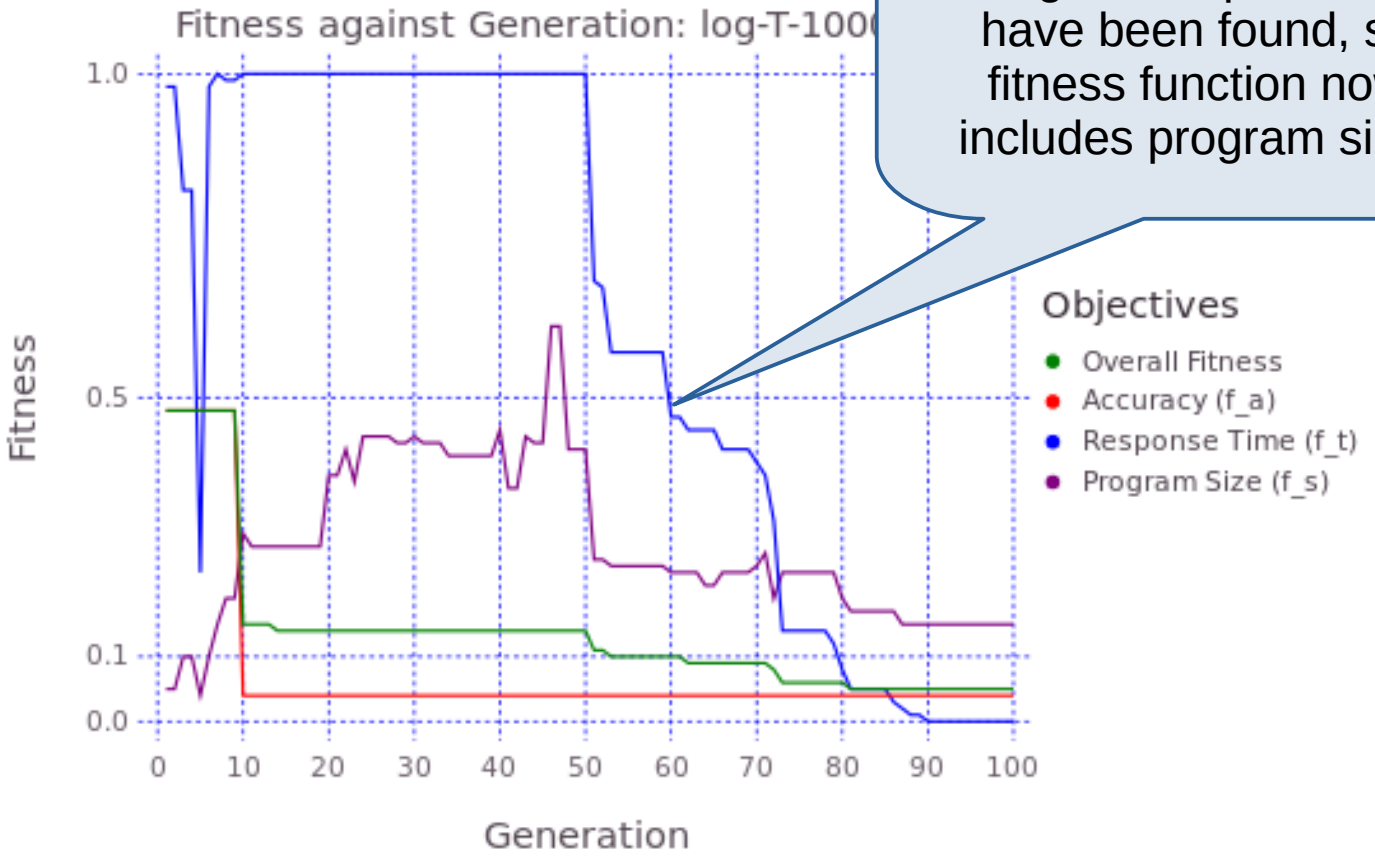
Phase 2 begins: Accurate model found, so fitness function now combines accuracy+response time

Phased Evolutio



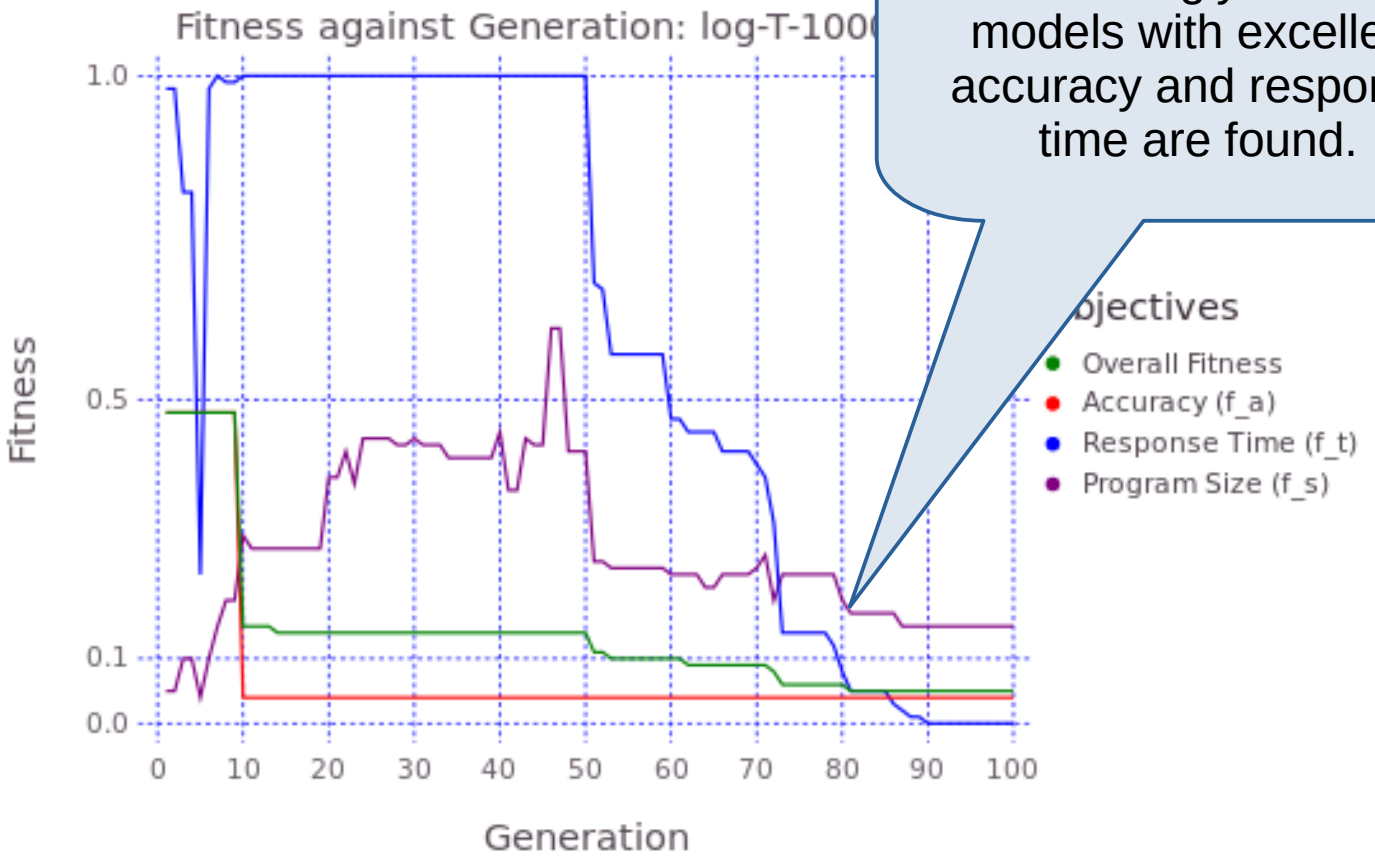
Within Phase 2: Program size continues to increase while seeking an accurate model with a good response time.

Phased Evolutio

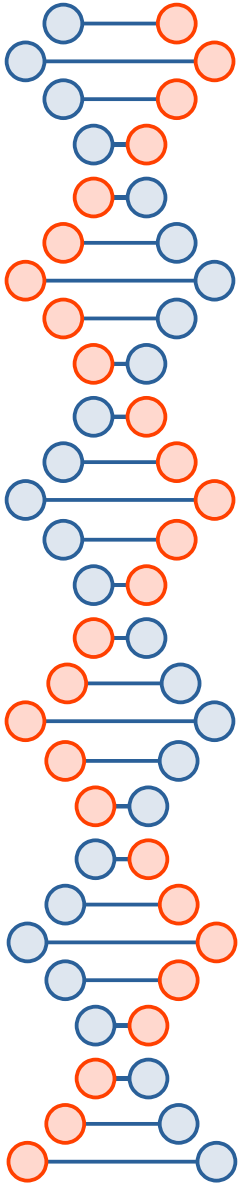


Phase 3 begins: Models with good response time have been found, so fitness function now includes program size.

Phased Evolutio

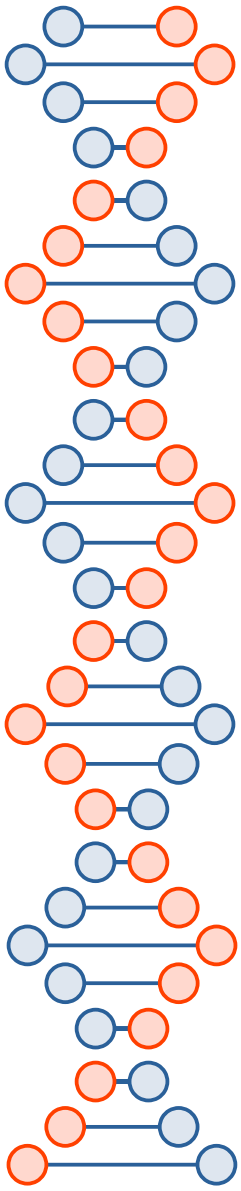


Within Phase 3: increasingly smaller models with excellent accuracy and response time are found.



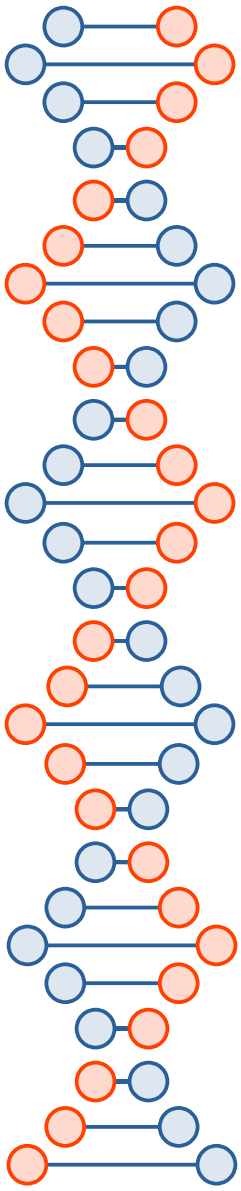
Part 3: Analysing the Output

- GEMS provides some tools to help analyse the process and output of the program synthesis process
- Logging: outputs information about the models on each cycle
- Post-processing: takes a population of models and rewrites/modifies them



Logging

```
;; Runs the GP system with given parameters, results logged to files.
(defun run-gp (&key (logger nil)) ; logger function
  (setf *phase* 1) ; initial phase for phased-evolution
  (gems:launch (operator-set) #'evaluate-program
               :total-generations *total-generations*
               :population-size *population-size*
               :initial-depth 1
               :maximum-depth 10
               :elitism t
               :type :steady-state
               :logger logger))
```



Logging

```
(defun run-group (name)
  (run-gp :logger (gems:combine-loggers
                   (gems:make-logger (format nil "log-~a.csv" name)
                                     :if-exists :supersede)
                   (gems:make-logger (format nil "population-~a.yml" name)
                                     :name name
                                     :kind :trace
                                     :filter #'(lambda (gen) (= gen *total-generations*))
                                     :if-exists :supersede
                                     ))))
```

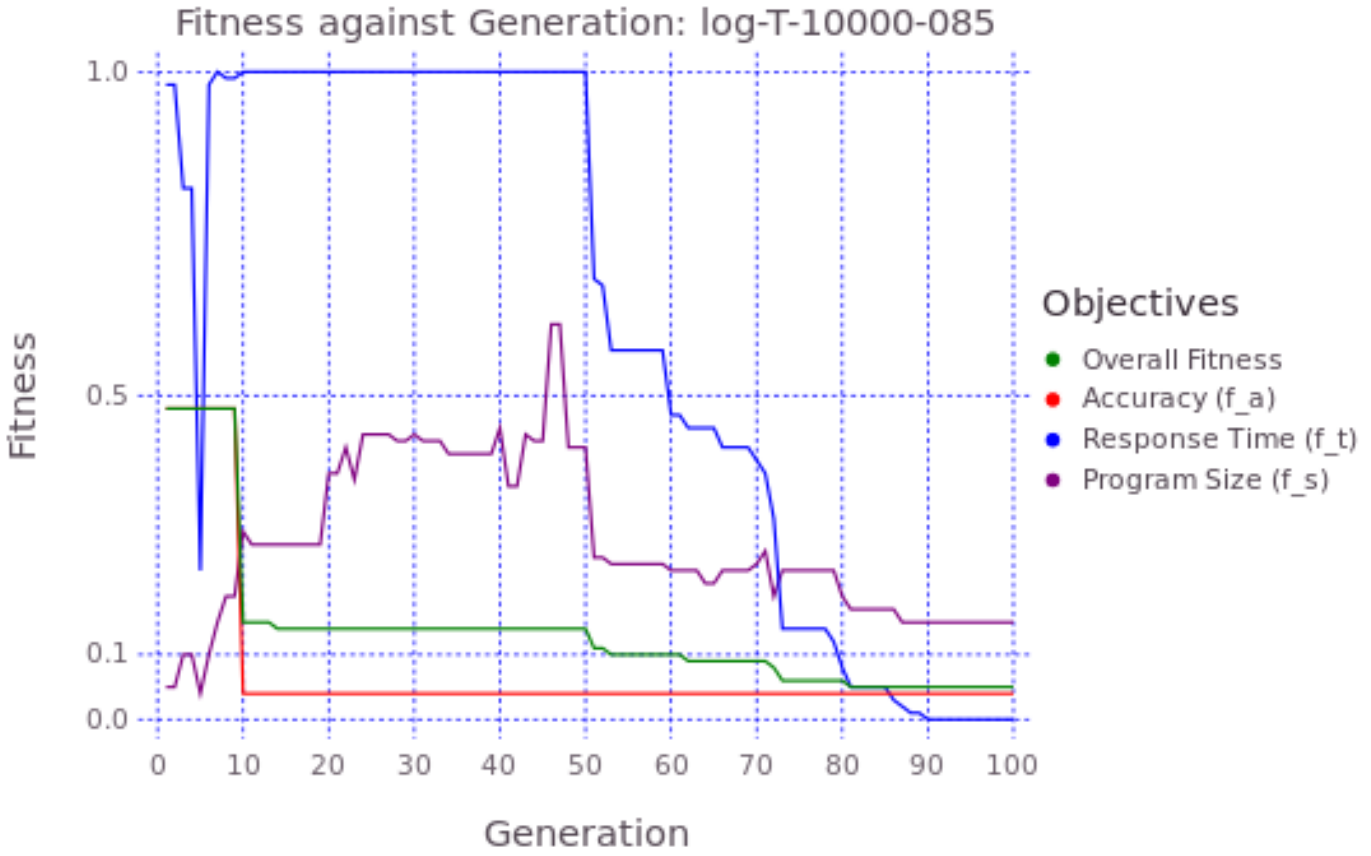
Trace loggers - produce whole population data
Short loggers - produce summary statistics per cycle

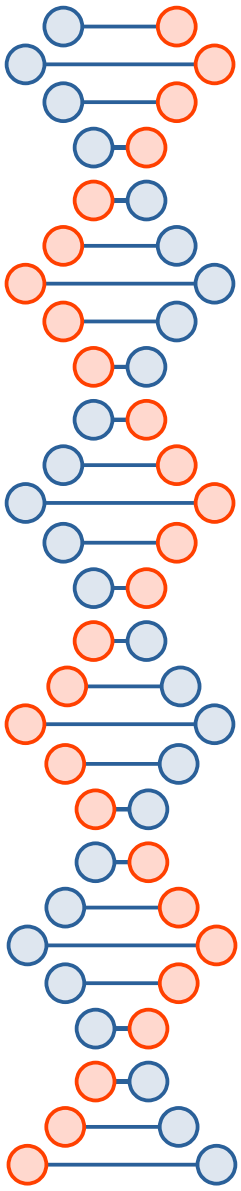
Logging

```
(defun evaluate-program (program)
  (let* ((results (run-experiment program))
         (accuracy (alexandria:mean (mapcar #'result-accuracy results)))
         (f-a (fitness-accuracy accuracy))
         (response-time (alexandria:mean (mapcar #'result-timing results)))
         (f-t (fitness-time response-time))
         (program-size (gems:program-size program))
         (f-s (fitness-size program-size)))
    (values  ; overall-fitness, optional extra information
      (overall-phased-fitness f-a f-t f-s)
      (list accuracy f-a response-time f-t program-size f-s *phase*) ; extra
      )))
```

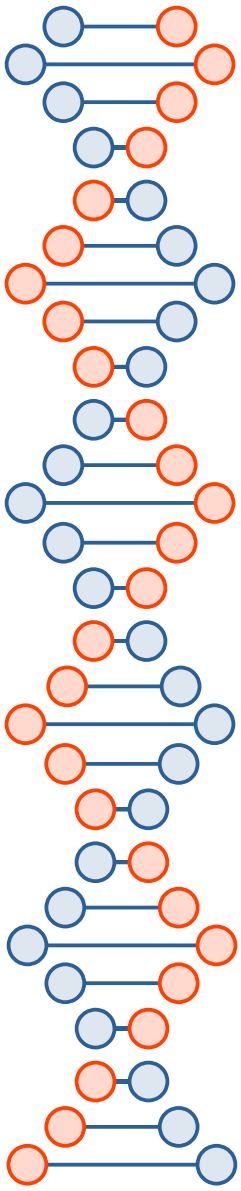
```
1, 0.478, 0.500, 0.478, 140.000, 0.387, 3.000, 0.015, 1.000
2, 0.478, 0.500, 0.478, 140.000, 0.387, 3.000, 0.015, 1.000
3, 0.478, 0.500, 0.478, 140.000, 0.387, 3.000, 0.015, 1.000
4, 0.478, 0.500, 0.478, 285.000, 0.304, 8.000, 0.040, 1.000
```

Logging


Fitness against Generation: log-T-10000-085
Fitness
1.0
0.5
0.1
0.0
0
10
20
30
40
50
60
70
80
90
100
Generation
Objectives
Overall Fitness
Accuracy (f_a)
Response Time (f_t)
Program Size (f_s)

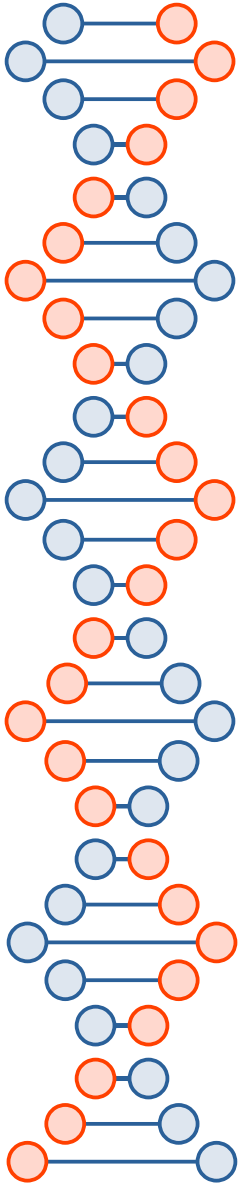


```
generation:
  number: 500
  individuals:
    - fitness: 0.056
      extras: (1 0.04493206 825 0.03779161 34 0.1683811 3)
      program: |
               (PROG2
                   (PROG4 (WAIT-200) (RESPOND-LEFT)
                    (PROG4
                     (PROG4 (WAIT-200) (RESPOND-RIGHT) (ACCESS-STM-1)
                      (ACCESS-STM-2))
                     (INPUT-TARGET) (WAIT-200)
                     (PROG4 (PUT-STM) (PROG2 (WAIT-200) (PUT-STM))
                      (PROG3 (RESPOND-RIGHT) (INPUT-RIGHT)
                       (IF (RESPOND-RIGHT)
                           (PUT-STM)
                           (INPUT-TARGET)))
                       (ACCESS-STM-1)))
                     (PUT-STM))
                   (PROG4 (ACCESS-STM-2) (ACCESS-STM-2)
                    (IF (COMPARE-1-3)
                        (ACCESS-STM-2)
                        (RESPOND-LEFT))
                    (COMPARE-1-3)))
    - fitness: 0.366
      extras: (1/2 0.47753397 630 0.08907235 27 0.13418579 3)
      program: |
               (PROG2
                   (PROG4 (RESPOND-LEFT) (RESPOND-LEFT)
                    (PROG4
```



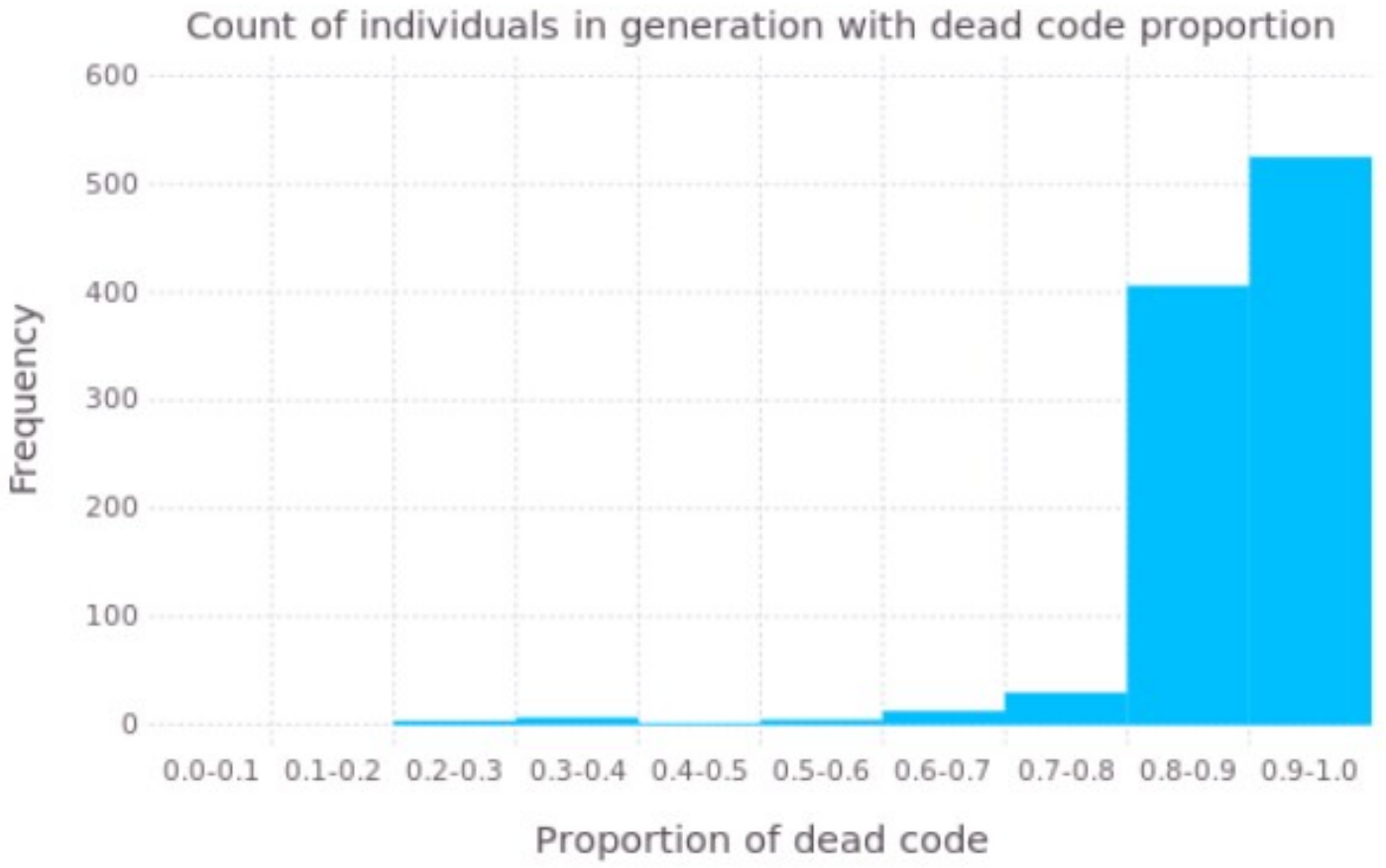
Post-Processing (1)

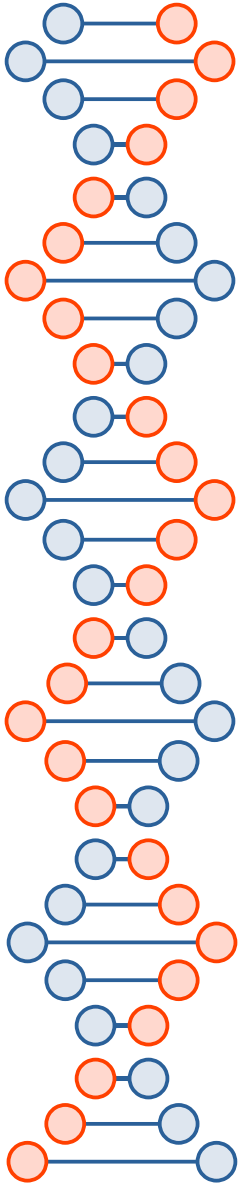
- The models generated by the GP system are typically messy:
  - contain bloat
  - operators duplicate or “mask” other operations
  - etc
- These obscure the understandability of the models and make it harder for a theorist to generate an explanation of the observed behaviour (of the models and humans).
- Post-processing rewrites the messy models into cleaner, semantically equivalent versions, suitable for analysis.



Post Processing (2)

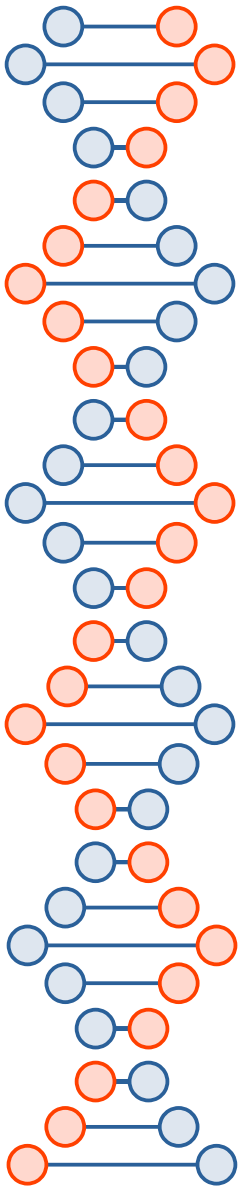
- Dead-code (“bloat”) is code which is never executed. This only arises in IF statements.
- Dead-code is identified by tracing the program and marking all nodes which are used. Unused branches can then be deleted.
  - (IF (CONDITION) (SOME-CODE) (UNUSED))
  - => (PROG2 (CONDITION) (SOME-CODE))
- (gems:clean-individuals programs #’run-experiment)


Count of individuals in generation with dead code proportion
Frequency
600
500
400
300
200
100
0
0.0-0.1
0.1-0.2
0.2-0.3
0.3-0.4
0.4-0.5
0.5-0.6
0.6-0.7
0.7-0.8
0.8-0.9
0.9-1.0
Proportion of dead code



Post-Processing (3)

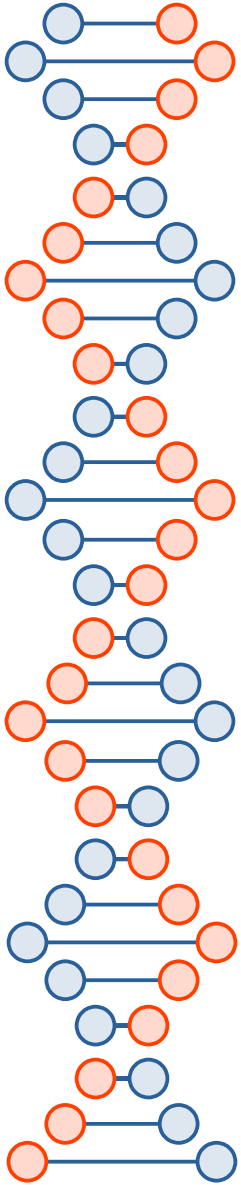
- Because the models are time-dependent, some operators are important only because they take up some processing time, e.g. waiting for an input to become available.
- These time-only operators do not have an effect on the output response, and some may be “masked” by later operators:
  - (PROG2 (INPUT-LEFT) (INPUT-RIGHT))
  - => (PROG2 (WAIT-INPUT) (INPUT-RIGHT))
- (clean-individuals-time ...) function not yet in library



Post-Processing Steps:

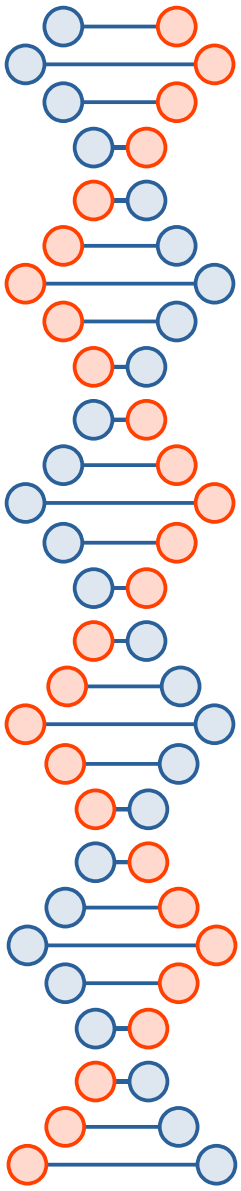
1. Read in the trace file
2. Extract models whose fitness is “good enough”
3. use gems:clean-individuals to remove dead code
4. use clean-individuals-time to remove time-only code
5. save / further process models

```
;; given a file containing a population of models
;; - perform post-processing
;; - return a list of models, in the form of gems:individual structures, so preserving fitness etc
(defun good-models (filename &optional (print-them nil))
  (let* ((models ; get models from final population
          (rest (first (last (gems:read-trace filename)))))
         (good-models ; extract those models within good-model threshold
          (remove-if #'(lambda (model) (> (gems:individual-fitness model) *good-model-threshold*))
                     models))
         (ndc-models ; remove dead-code from the model programs
          (gems:clean-individuals good-models #'run-experiment))
         (nto-models ; remove time-only code from the model programs
          (clean-individuals-time ndc-models #'run-experiment)))
```



Post Processing (4)

- GP system run six times with 500 individuals over 2000 generations
- 1164 distinct “good” models - overall fitness < 0.1
  - this is over a third of the total - good models in GP tend to proliferate in the population, with minor variations
- deleting dead-code (“bloat”) reduces this to 248 models
- rewriting time-only operators reduces this to 11 models
  - a 99% reduction in total number of models



Model Similarity (1)

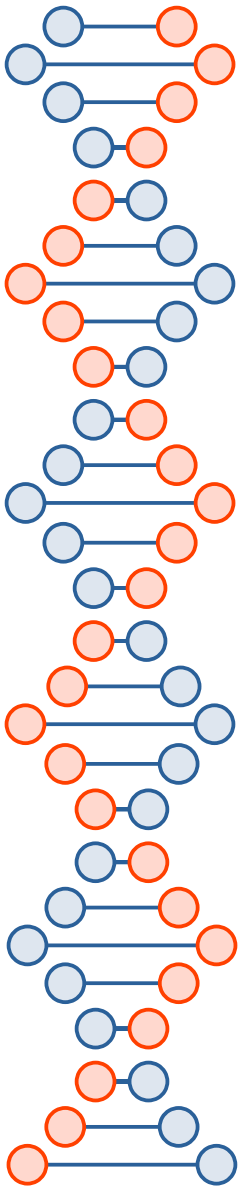
(IF (ACCESS-1)
(PROG2 (INPUT-RIGHT) (INPUT-LEFT))
(INPUT-TARGET))

(IF (ACCESS-2)
(PROG2 (WAIT-25) (INPUT-RIGHT))
(INPUT-TARGET))

$$\frac{|A \cap B|}{|A \cup B|}$$

Components 1 (A):
(IF ACCESS-1 PROG2 INPUT-TARGET)
(PROG2 INPUT-RIGHT INPUT-LEFT)
IF ACCESS-1 PROG2 INPUT-RIGHT INPUT-LEFT INPUT-TARGET

Components 2 (B):
(IF ACCESS-2 PROG2 INPUT-TARGET)
(PROG2 WAIT-25 INPUT-RIGHT)
IF ACCESS-1 PROG2 INPUT-RIGHT INPUT-TARGET WAIT-25



Model Similarity (2)

(IF (ACCESS-1)
(PROG2 (INPUT-RIGHT) (INPUT-LEFT))
(INPUT-TARGET))

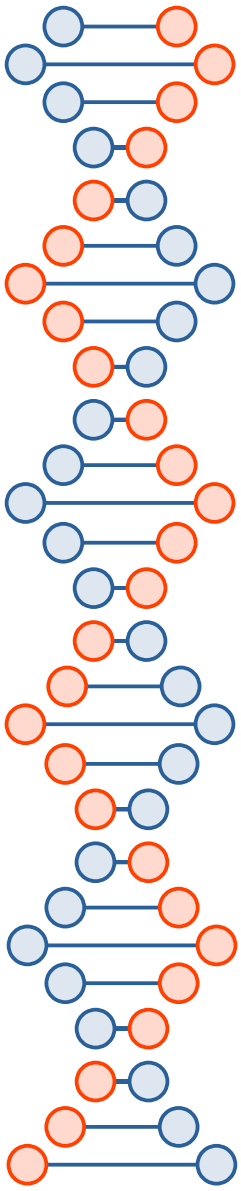
(IF (ACCESS-2)
(PROG2 (WAIT-25) (INPUT-RIGHT))
(INPUT-TARGET))

$$\frac{|A \cap B|}{|A \cup B|}$$

Components 1 (A):
(IF ACCESS-1 PROG2 INPUT-TARGET)
(PROG2 INPUT-RIGHT INPUT-LEFT)
IF ACCESS-1 PROG2 INPUT-RIGHT INPUT-LEFT INPUT-TARGET

Similarity: 5 / 11 = 0.45

Components 2 (B):
(IF ACCESS-2 PROG2 INPUT-TARGET)
(PROG2 WAIT-25 INPUT-RIGHT)
IF ACCESS-1 PROG2 INPUT-RIGHT INPUT-TARGET WAIT-25



Similarity Data

- (gems:program-similarity program-1 program-2)
  - computes the similarity of two programs
- (gems:write-similarity-individuals programs filename)
  - writes similarities in following (GNUplot) format

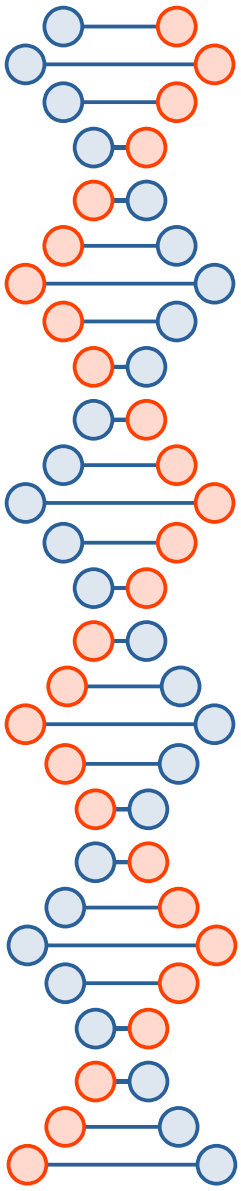
0 0 1.00

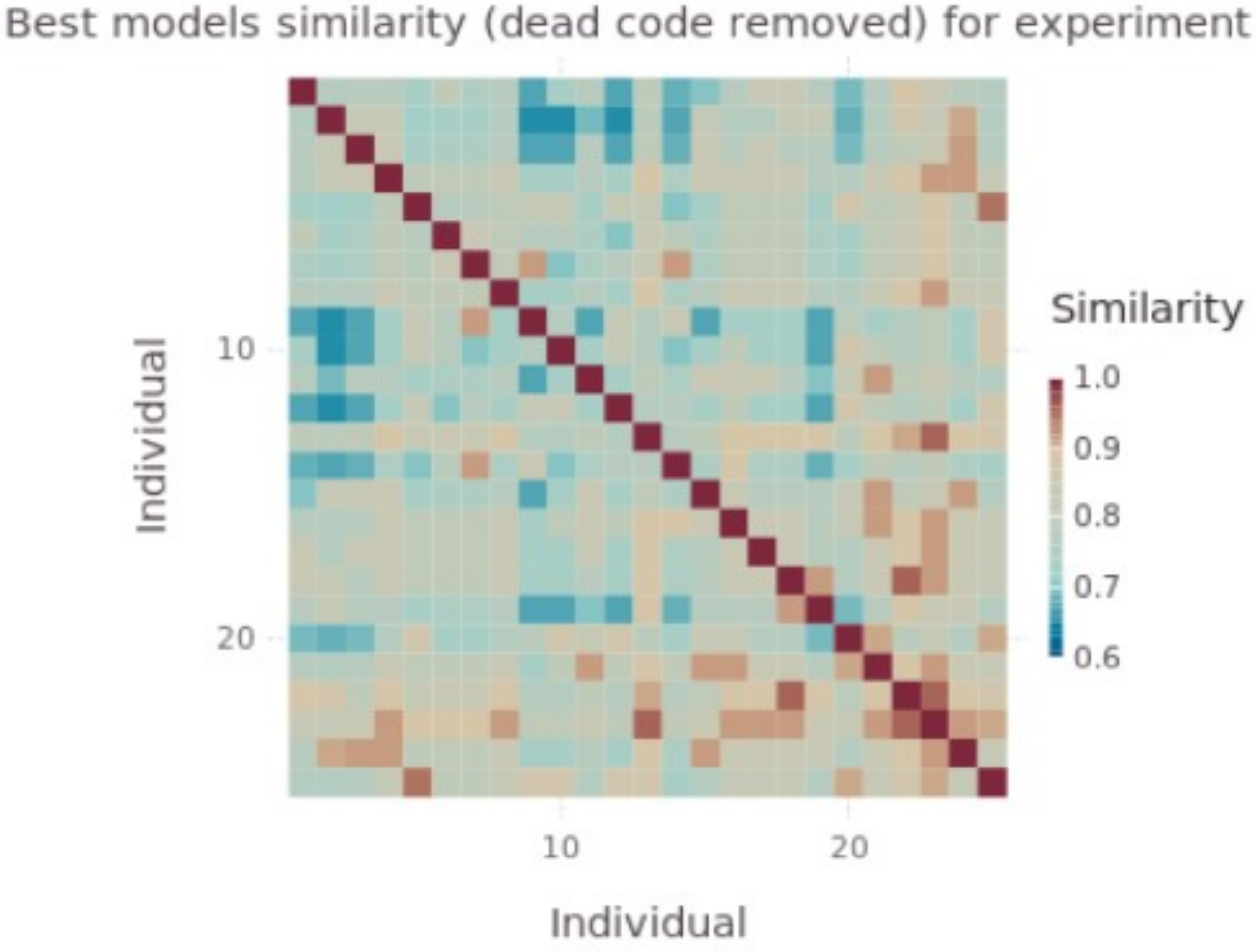
...

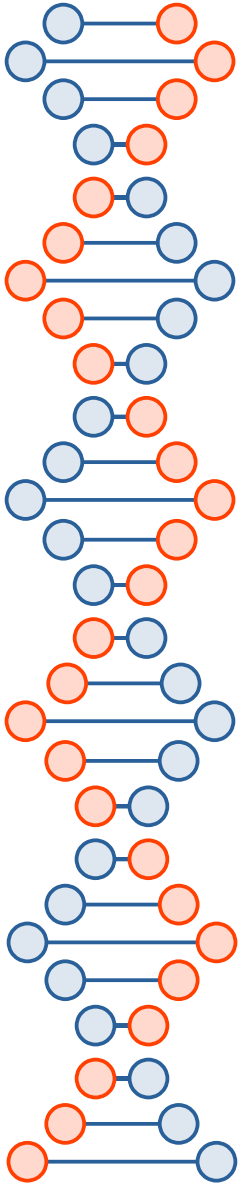
0 9 0.14

1 0 0.00

...




Best models similarity (dead code removed) for experiment 2
Individual
10
20
10
20
Individual
Similarity
1.0
0.9
0.8
0.7
0.6



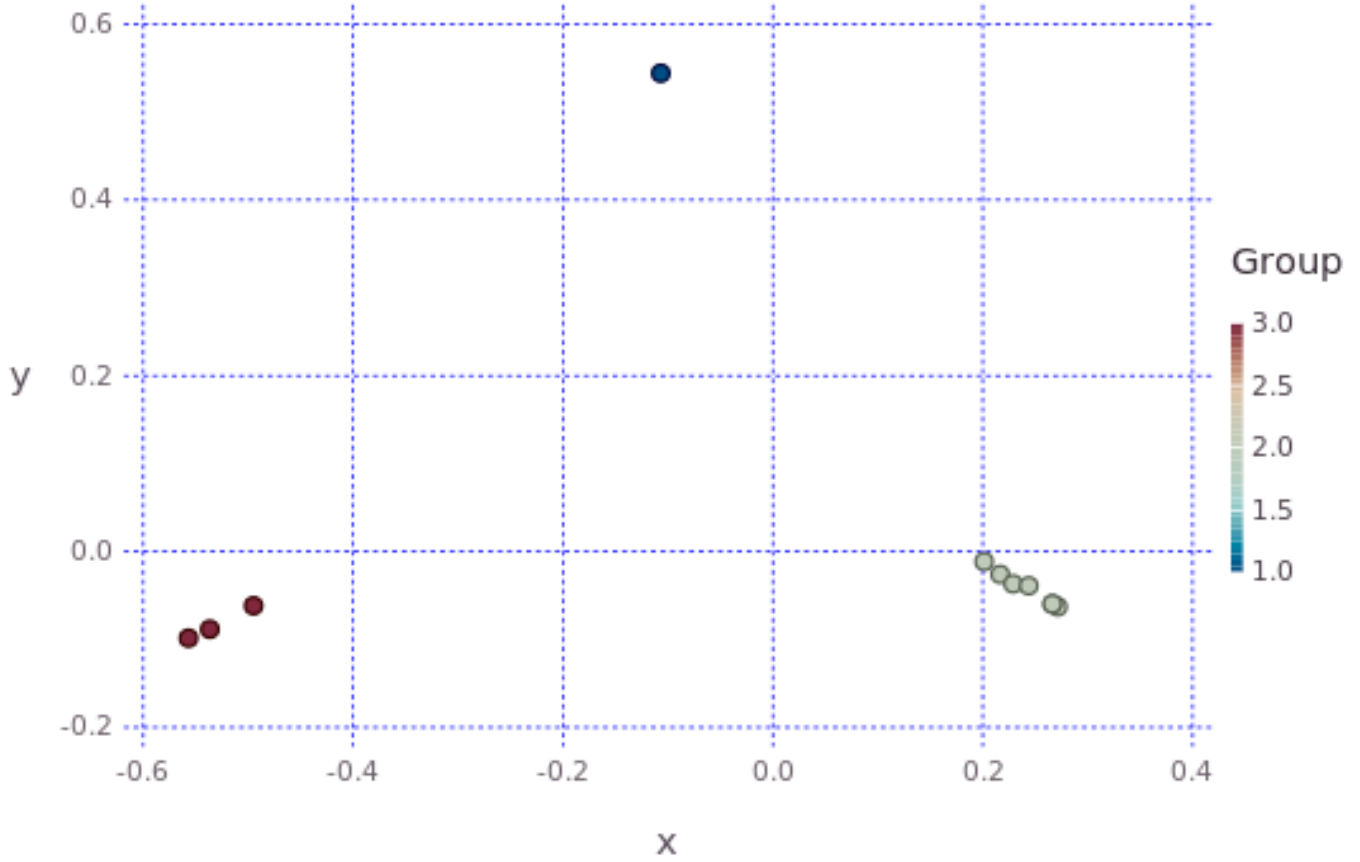
Multi-Dimensional Scaling

Takes the similarity between pairs of models and creates 2D (or n-D) coordinates where the distance between each pair of points reflects the similarity between the models.

- Scatter plot visualisations
- Analysis of models in groups using Clustering

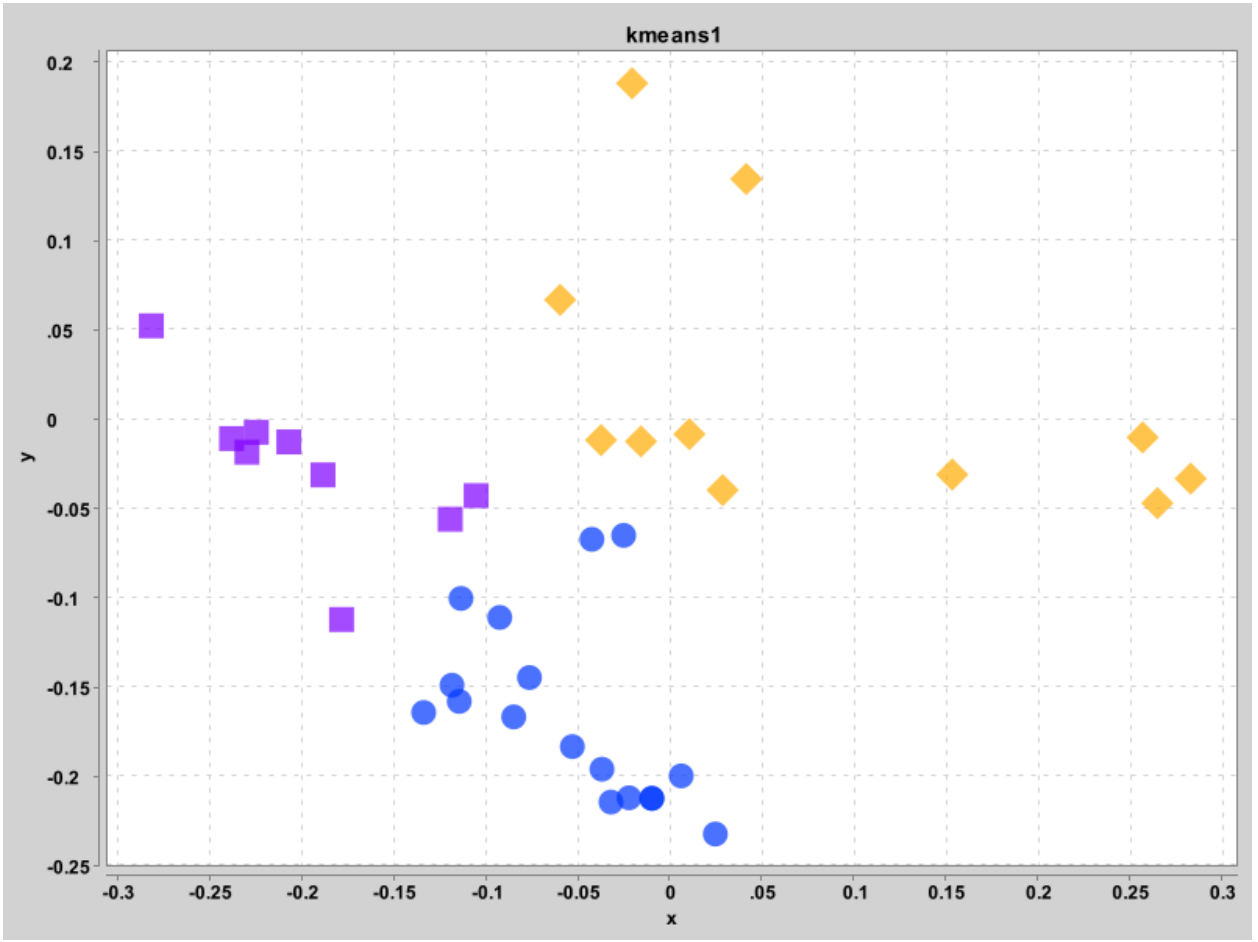
External tool used for this, e.g. Julia or a Java library.

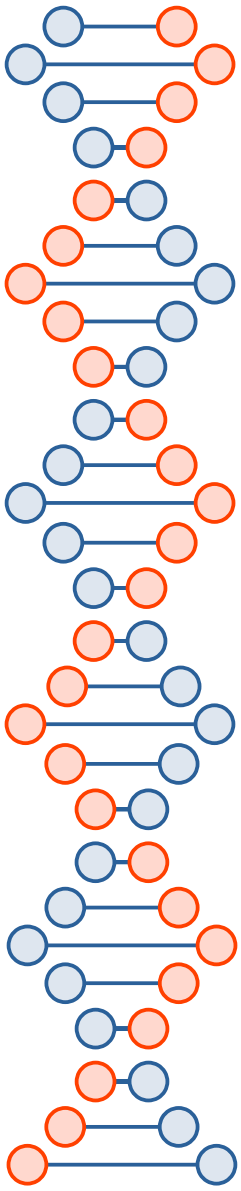
Scatter Plot of Model Similarity



Clustering

- k = 3





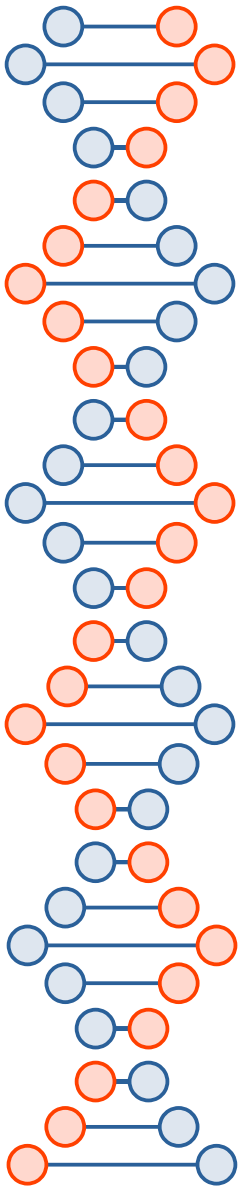
```
if target is visible:
  set model 'current' to target
wait for 140ms
push model 'current' onto top of STM
loop 3 times:
  loop 5 times:
    if stimuli are visible:
      set model 'current' to left input
if stimuli are visible:
  set model 'response' to "R"
push model 'current' onto top of STM
if first item in STM equals second item:
  set model 'current' to 1
else:
  set model 'current' to 0
if model 'current' is 1:
  if stimuli are visible:
    set model 'response' to "L"
else:
  wait for 70ms
wait for 70ms
```

Looks for Target

Waits for Input

Does Comparison

Outputs Response



```
loop 2 times do
  if target is visible then
    set model 'current' to target
  end
  loop 2 times do
    wait for 100ms
    push model 'current' onto top of STM
    wait for 50ms
    if stimuli are visible then
      set model 'current' to right input
    end
  end
  if first item in STM equals third item then
    set model 'current' to 1
  else
    set model 'current' to 0
  end
...
```

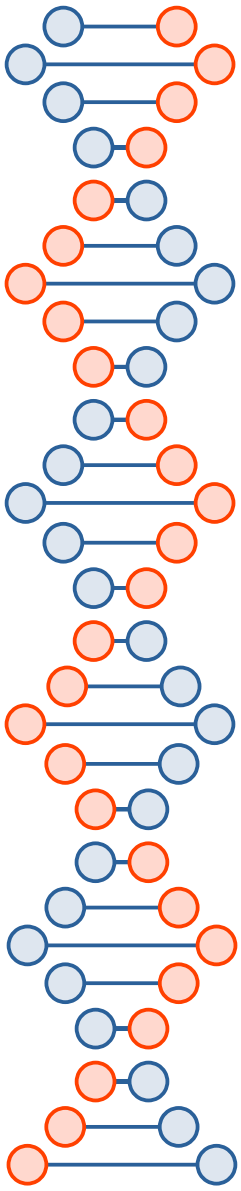
Big loop:

Looks for Target

Looks for Input

Does Comparison

Outputs Response



More Information

There is a related talk in Session 2b (15:00, Wednesday) of the conference: A. Pirrone, P.C.R. Lane, L.K. Bartlett, N.Javed and F. Gobet, 'Heuristic search of heuristics'

- GEMS: https://gems-science.netlify.app
- GEMS software: https://gems-science.netlify.app/software
- GEMS software repository: https://notabug.org/gems
- Peter Lane: https://go.herts.ac.uk/peter-lane