Example of a run

```
May 19 17:05 1987  trace3.31 Page 1


(config 31 3 5 24 3 14)
Le jeu des chiffres
Initial configuration :
   Target : 31
   Bricks : 3 5 24 3 14
Node cyto-target created
Node cyto-brick1 created
Node cyto-brick2 created
Node cyto-brick4 created
Node cyto-brick5 created
About to post codelet look-for-blx (30 3 10)
About to post codelet look-for-blx (30 5 6)
Node cyto-brick3 created
Node cyto-block9-v1 created
Node times3-3-v1 created
Node times3-3-v1 killed
Node cyto-block9-v1 killed
About to post codelet look-for-blx (30 3 10)
About to post codelet look-for-blx (30 5 6)
Node cyto-block27-v2 created
Node plus3-24-v2 created
Node cyto-target-4-v3 created
Node plus27-4-v3 created
Node cyto-target-1-v4 created
Node plus4-1-v4 created
;; gc: flonum +25 (135)
Node cyto-target-2-v5 created
Node plus1-2-v5 created
About to post codelet look-for-diff (0)
Node plus1-2-v5 killed
Node cyto-target-2-v5 killed
Node plus4-1-v4 killed
Node cyto-target-1-v4 killed
Node cyto-target-1-v6 created
Node plus4-1-v6 created
Node plus4-1-v6 killed
Node cyto-target-1-v6 killed
Node cyto-target-1-v7 created
Node plus4-1-v7 created
Node plus27-4-v3 killed
Node cyto-target-4-v3 killed
Node plus4-1-v7 killed
Node cyto-target-1-v7 killed
Node plus3-24-v2 killed
Node cyto-block27-v2 killed
Node cyto-target-7-v8 created
Node plus24-7-v8 created
Node cyto-target-7-v9 created
Node plus7-7-v9 created
Node cyto-target-2-v10 created
Node plus5-2-v10 created
Node plus5-2-v10 killed
Node cyto-target-2-v10 killed
Node cyto-block2-v11 created
Node plus2-3-v11 created
Node cyto-target-5-v12 created
```

```
May 19 17:05 1987  trace3.31 Page 2


Node plus2-5-v12 created
Node cyto-target-2-v13 created
Node plus3-2-v13 created
```

```
(config 31 3 5 24 3 14)
Le jeu des chiffres
Initial configuration :
   Target : 31
   Bricks : 3 5 24 3 14
Node cyto-target created
Node cyto-brick1 created
Node cyto-brick2 created
Node cyto-brick4 created
Node cyto-brick5 created
About to post codelet look-for-blx (30 3 10)
About to post codelet look-for-blx (30 5 6)
Node cyto-brick3 created
Node cyto-block9-v1 created
Node times3-3-v1 created
Node times3-3-v1 killed
Node cyto-block9-v1 killed
About to post codelet look-for-blx (30 3 10)
About to post codelet look-for-blx (30 5 6)
Node cyto-block27-v2 created
Node plus3-24-v2 created
Node cyto-target-4-v3 created
Node plus27-4-v3 created
Node cyto-target-1-v4 created
Node plus4-1-v4 created
;; gc: flonum +25 (135)
Node cyto-target-2-v5 created
Node plus1-2-v5 created
About to post codelet look-for-diff (0)
Node plus1-2-v5 killed
Node cyto-target-2-v5 killed
Node plus4-1-v4 killed
Node cyto-target-1-v4 killed
Node cyto-target-1-v6 created
Node plus4-1-v6 created
Node plus4-1-v6 killed
Node cyto-target-1-v6 killed
Node cyto-target-1-v7 created
Node plus4-1-v7 created
Node plus27-4-v3 killed
Node cyto-target-4-v3 killed
Node plus4-1-v7 killed
Node cyto-target-1-v7 killed
Node plus3-24-v2 killed
Node cyto-block27-v2 killed
Node cyto-target-7-v8 created
Node plus24-7-v8 created
Node cyto-target-7-v9 created
Node plus7-7-v9 created
Node cyto-target-2-v10 created
Node plus5-2-v10 created
Node plus5-2-v10 killed
Node cyto-target-2-v10 killed
Node cyto-block2-v11 created
Node plus2-3-v11 created
Node cyto-target-5-v12 created
```



```
Node plus2-5-v12 created
Node cyto-target-2-v13 created
Node plus3-2-v13 created
```

```
; PNODE FLAVOR
(defflavor pnode
           (activation
            spreadable-activation
            temp-activation-holder
            neighbors
            instances
            name
            type
            value
            codelets
            short-name ; for graphics
            x-region
            y-region
            x-box
            y-box
            x-act
            y-act
            x-name
            y-name
            old-boxsize)

           ()
; Pnet node       - activation is the level of activation
;                 - spreadable-activation is the level of activation times a
;                                                    decay factor
;                 - temp-activation-holder is a temporary storage of activation
;                                                    received
;                 - neigbors are the neighbors and the correspond type of link
;                 - instances are the instances in the cyto
;                 - name is the full name of the node
;                 - type is the type of the node (num,op,other)
;                 - value is the possible value associated with the node
;                 - codelets are the codelets attached to the node
           :settable-instance-variables
           :inittable-instance-variables
           :gettable-instance-variables)

;INIT PNET
(defun init-pnet ()
; initializes the pnet.
  (setq node-1 (make-instance 'pnode
                 :value 1
                 :neighbors
                    '((plus1-1 result+)
                      (plus1-2 result+)
                      (plus1-3 result+)
                      (plus1-4 result+)
                      (plus1-5 result+)
                      (plus1-6 result+)
                      (plus1-7 result+)
                      (plus1-8 result+)
                      (plus1-9 result+))
                 :name 'one
                 :short-name "1"
```

```
                    :type 'num))
    (setq node-2 (make-instance 'pnode
                    :value 2
                    :neighbors
                       '((plus1-2 result+)
                         (plus2-2 result+)
                         (plus2-3 result+)
                         (plus2-4 result+)
                         (plus2-5 result+)
                         (plus2-6 result+)
                         (plus2-7 result+)
                         (plus2-8 result+)
                         (times2-2 operand)
                         (times2-3 operand)
                         (times2-4 operand)
                         (times2-5 operand)
                         (times2-7 operand)
                         (times2-10 operand)
                         (times2-20 operand)
                         (times2-12 operand)
                         (plus1-1 result+))
                    :name 'two
                    :short-name "2"
                    :type 'num))
     (setq node-3 (make-instance 'pnode
                    :value 3
                    :neighbors
                       '((plus1-3 result+)
                         (plus2-3 result+)
                         (plus3-3 result+)
                         (plus3-4 result+)
                         (plus3-5 result+)
                         (plus3-6 result+)
                         (plus3-7 operand)
                         (times2-3 operand)
                         (times3-3 operand)
                         (times3-10 operand)
                         (times3-20 operand)
                         (plus1-2 result+)
                         (node-4 similar))
                    :name 'three
                    :short-name "3"
                    :type 'num))
     (setq node-4 (make-instance 'pnode
                    :value 4
                    :neighbors
                       '((plus1-4 result+)
                         (plus2-4 result+)
                         (plus3-4 result+)
                         (plus4-4 result+)
                         (plus4-5 result+)
                         (plus4-6 result+)
                         (times2-4 operand)
                         (times4-4 operand)
                         (times4-10 operand)
                         (times4-20 operand)
```

```
                    (plus1-3 result+)
                    (plus2-2 result+)
                    (times2-2 resultx)
                    (node-3 similar)
                    (node-5 similar))
                :name 'four
                :short-name "4"
                :type 'num))

(setq node-5 (make-instance 'pnode
                :value 5
                :neighbors
                   '((plus1-5 result+)
                     (plus2-5 result+)
                     (plus3-5 result+)
                     (plus4-5 result+)
                     (plus5-5 result+)
                     (plus5-10 result+)
                     (times2-5 operand)
                     (times5-5 operand)
                     (times5-6 operand)
                     (times5-10 operand)
                     (times5-20 operand)
                     (plus1-4 result+)
                     (plus2-3 result+)
                     (node-4 similar)
                     (node-6 similar))
                :name 'five
                :short-name "5"
                :type 'num))

(setq node-6 (make-instance 'pnode
                :value 6
                :neighbors
                   '((plus1-6 result+)
                     (plus2-6 result+)
                     (plus3-6 result+)
                     (plus4-6 result+)
                     (plus1-5 result+)
                     (plus2-4 result+)
                     (plus3-3 result+)
                     (times2-3 resultx)
                     (times5-6 operand)
                     (times6-10 operand)
                     (node-5 similar)
                     (node-7 similar))
                :name 'six
                :short-name "6"
                :type 'num))

(setq node-7 (make-instance 'pnode
                :value 7
                :neighbors
                   '((plus1-7 result+)
                     (plus2-7 result+)
                     (plus3-7 result+)
```

```
                    (plus7-8 result+)
                    (times2-7 operand)
                    (times7-7 operand)
                    (times7-10 operand)
                    (plus1-6 result+)
                    (plus2-5 result+)
                    (plus3-4 result+)
                    (node-6 similar)
                    (node-8 similar))
                :name 'seven
                :short-name "7"
                :type 'num))

(setq node-8 (make-instance 'pnode
                :value 8
                :neighbors
                   '((plus1-8 result+)
                     (plus2-8 result+)
                     (plus1-7 result+)
                     (plus2-6 result+)
                     (plus3-5 result+)
                     (plus4-4 result+)
                     (plus7-8 result+)
                     (times2-4 result+)
                     (times8-10 operand)
                     (node-7 similar)
                     (node-9 similar))
                 :name 'eight
                 :short-name "8"
                 :type 'num))

 (setq node-9 (make-instance 'pnode
                 :value 9
                 :neighbors
                    '((plus1-9 result+)
                      (times9-9 operand)
                      (times9-10 operand)
                      (plus1-8 result+)
                      (plus2-7 result+)
                      (plus3-6 result+)
                      (plus4-5 result+)
                      (times3-3 resultx)
                      (node-8 similar)
                      (node-10 similar))
                  :name 'nine
                  :short-name "9"
                  :type 'num))

  (setq node-10 (make-instance 'pnode
                  :value 10
                  :neighbors
                     '((times2-10 operand)
                       (times3-10 operand)
                       (times4-10 operand)
                       (times5-10 operand)
                       (times6-10 operand)
```

```
                    (times7-10 operand)
                    (times8-10 operand)
                    (times9-10 operand)
                    (times10-10 operand)
                    (times10-15 operand)
                    (plus1-9 result+)
                    (plus2-8 result+)
                    (plus3-7 result+)
                    (plus4-6 result+)
                    (plus5-5 result+)
                    (plus5-10 result+)
                    (times2-5 resultx)
                    (node-9 similar))
                :name 'ten
                :short-name "10"
                :type 'num))

(setq node-12 (make-instance 'pnode
                :value 12
                :neighbors
                   '((times2-12 operand)
                     (node-15 similar))
                :name 'twelfe
                :short-name "12"
                :type 'num))

(setq node-15 (make-instance 'pnode
                :value 15
                :neighbors
                   '((plus7-8 result+)
                     (plus5-10 result+)
                     (times2-7 similar)
                     (times10-15 operand)
                     (node-12 similar)
                     (node-16 similar)
                     (node-20 similar))
                :name 'fifteen
                :short-name "15"
                :type 'num))

(setq node-16 (make-instance 'pnode
                :value 16
                :neighbors
                   '((times4-4 resultx))
                :name 'sixteen
                :short-name "16"
                :type 'num))

(setq node-20 (make-instance 'pnode
                :value 20
                :neighbors
                   '((times2-20 operand)
                     (times3-20 operand)
                     (times4-20 operand)
                     (times5-20 operand)
                     (times2-10 operand)
```

```
                    (node-15 similar)
                    (node-25 similar))
                :name 'twenty
                :short-name "20"
                :type 'num))

(setq node-25 (make-instance 'pnode
                :value 25
                :neighbors
                   '((times5-5 resultx)
                     (times2-12 similar)
                     (node-20 similar)
                     (node-30 similar))
                :name 'twenty-five
                :short-name "25"
                :type 'num))

(setq node-30 (make-instance 'pnode
                :value 30
                :neighbors
                   '((times3-10 resultx)
                     (times5-6 resultx)
                     (node-25 similar)
                     (node-40 similar))
                :name 'thirty
                :short-name "30"
                :type 'num))

(setq node-40 (make-instance 'pnode
                :value 40
                :neighbors
                   '((times4-10 resultx)
                     (times2-20 resultx)
                     (node-30 similar)
                     (node-50 similar))
                :name 'fourty
                :short-name "40"
                :type 'num))

(setq node-50 (make-instance 'pnode
                :value 50
                :neighbors
                   '((times5-10 resultx)
                     (times7-7 similar)
                     (node-40 similar)
                     (node-60 similar))
                :name 'fifty
                :short-name "50"
                :type 'num))

(setq node-60 (make-instance 'pnode
                :value 60
                :neighbors
                   '((times6-10 resultx)
                     (times3-20 resultx)
                     (node-50 similar)
```

```
                    (node-70 similar))
                :name 'sixty
                :short-name "60"
                :type 'num))

(setq node-70 (make-instance 'pnode
                :value 70
                :neighbors
                   '((times7-10 resultx)
                     (node-60 similar)
                     (node-80 similar))
                :name 'seventy
                :short-name "70"
                :type 'num))

(setq node-80 (make-instance 'pnode
                :value 80
                :neighbors
                   '((times8-10 resultx)
                     (times4-20 resultx)
                     (node-70 similar)
                     (node-81 similar)
                     (node-90 similar))
                :name 'eighty
                :short-name "80"
                :type 'num))

(setq node-81 (make-instance 'pnode
                :value 81
                :neighbors
                   '((times9-9 resultx)
                     (node-80 similar))
                :name 'eighty-one
                :short-name "81"
                :type 'num))

(setq node-90 (make-instance 'pnode
                :value 90
                :neighbors
                   '((times9-10 resultx)
                     (node-80 similar)
                     (node-100 similar))
                :name 'ninety
                :short-name "90"
                :type 'num))

(setq node-100 (make-instance 'pnode
                 :value 100
                 :neighbors
                    '((times10-10 resultx)
                      (times5-20 resultx)
                      (node-90 similar)
                      (node-150 similar))
                 :name 'one-hundred
                 :short-name "100"
                 :type 'num))
```

```
(setq node-150 (make-instance 'pnode
                :value 150
                :neighbors
                   '((times10-15 resultx)
                     (node-100 similar))
                :name 'one-hundred-fifty
                :short-name "150"
                :type 'num))

(setq node-mul10 (make-instance 'pnode
                  :neighbors
                     '((node-10 instance))
                  :name 'multiple-of-ten
                  :short-name "mul10"))

(setq node-multiply (make-instance 'pnode
                     :neighbors
                        '((node-add operation)
                          (node-subtract operation))
                     :short-name "mult"
                     :name 'multiply))

(setq node-add (make-instance 'pnode
                :neighbors
                   '((node-multiply operation)
                     (node-subtract operation))
                :short-name "add"
                :name 'add))

(setq node-subtract (make-instance 'pnode
                     :neighbors
                        '((node-multiply operation)
                          (node-add operation))
                     :short-name "sub"
                     :name 'subtraction
                     :codelets
                        '((look-for-diff %first-threshold%
                            %first-urgency% (0)))))


(setq plus1-1 (make-instance 'pnode
               :neighbors
                  '((node-1 result+)
                    (node-2 result+))
               :name 'plus1-1
               :short-name "1+1"
               :codelets
                  '((look-for-bl+ %first-threshold%
                      %second-urgency% (2 1 1)))))

(setq plus1-2 (make-instance 'pnode
               :neighbors
                  '((node-1 result+)
                    (node-2 result+)
                    (node-3 result+))
```

```
                    :name 'plus1-2
                    :short-name "1+2"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                             %second-urgency% (3 1 2)))))

(setq plus1-3 (make-instance 'pnode
                    :neighbors
                       '((node-1 result+)
                         (node-3 result+)
                         (node-4 result+))
                    :name 'plus1-3
                    :short-name "1+3"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                             %second-urgency% (4 1 3)))))

(setq plus1-4 (make-instance 'pnode
                    :neighbors
                       '((node-1 result+)
                         (node-4 result+)
                         (node-5 result+))
                    :name 'plus1-4
                    :short-name "1+4"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                             %second-urgency% (5 1 4)))))

(setq plus1-5 (make-instance 'pnode
                    :neighbors
                       '((node-1 result+)
                         (node-5 result+)
                         (node-6 result+))
                    :name 'plus1-5
                    :short-name "1+5"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                             %second-urgency% (6 1 5)))))

(setq plus1-6 (make-instance 'pnode
                    :neighbors
                       '((node-1 result+)
                         (node-6 result+)
                         (node-7 result+))
                    :name 'plus1-6
                    :short-name "1+6"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                             %second-urgency% (7 1 6)))))

(setq plus1-7 (make-instance 'pnode
                    :neighbors
                       '((node-1 result+)
                         (node-7 result+)
                         (node-8 result+))
                    :name 'plus1-7
```

```
                :short-name "1+7"
                :codelets
                   '((look-for-bl+ %first-threshold%
                         %second-urgency% (8 1 7)))))


(setq plus1-8 (make-instance 'pnode
                :neighbors
                   '((node-1 result+)
                     (node-8 result+)
                     (node-9 result+))
                :name 'plus1-8
                :short-name "1+8"
                :codelets
                   '((look-for-bl+ %first-threshold%
                         %second-urgency% (9 1 8)))))

(setq plus1-9 (make-instance 'pnode
                :neighbors
                   '((node-1 result+)
                     (node-9 result+)
                     (node-10 result+))
                :name 'plus1-9
                :short-name "1+9"
                :codelets
                   '((look-for-bl+ %first-threshold%
                         %second-urgency% (10 1 9)))))

(setq plus2-2 (make-instance 'pnode
                :neighbors
                   '((node-2 result+)
                     (node-4 result+))
                :name 'plus2-2
                :short-name "2+2"
                :codelets
                   '((look-for-bl+ %first-threshold%
                         %second-urgency% (4 2 2)))))

(setq plus2-3 (make-instance 'pnode
                :neighbors
                   '((node-2 result+)
                     (node-3 result+)
                     (node-5 result+))
                :name 'plus2-3
                :short-name "2+3"
                :codelets
                   '((look-for-bl+ %first-threshold%
                         %second-urgency% (5 2 3)))))

(setq plus2-4 (make-instance 'pnode
                :neighbors
                   '((node-2 result+)
                     (node-4 result+)
                     (node-6 result+))
                :name 'plus2-4
```

```
                    :short-name "2+4"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                              %second-urgency% (6 2 4)))))

(setq plus2-5 (make-instance 'pnode
                    :neighbors
                       '((node-2 result+)
                         (node-5 result+)
                         (node-7 result+))
                    :name 'plus2-5
                    :short-name "2+5"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                              %second-urgency% (7 2 5)))))

(setq plus2-6 (make-instance 'pnode
                    :neighbors
                       '((node-2 result+)
                         (node-6 result+)
                         (node-8 result+))
                    :name 'plus2-6
                    :short-name "2+6"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                              %second-urgency% (8 2 6)))))

(setq plus2-7 (make-instance 'pnode
                    :neighbors
                       '((node-2 result+)
                         (node-7 result+)
                         (node-9 result+))
                    :name 'plus2-7
                    :short-name "2+7"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                              %second-urgency% (9 2 7)))))

(setq plus2-8 (make-instance 'pnode
                    :neighbors
                       '((node-2 result+)
                         (node-8 result+)
                         (node-10 result+))
                    :name 'plus2-8
                    :short-name "2+8"
                    :codelets
                       '((look-for-bl+ %first-threshold%
                              %second-urgency% (10 2 8)))))

(setq plus3-3 (make-instance 'pnode
                    :neighbors
                       '((node-3 result+)
                         (node-6 result+))
                    :name 'plus3-3
                    :short-name "3+3"
                    :codelets
```

```
                '((look-for-bl+ %first-threshold%
                        %second-urgency% (6 3 3)))))

(setq plus3-4 (make-instance 'pnode
              :neighbors
                 '((node-3 result+)
                   (node-4 result+)
                   (node-7 result+))
              :name 'plus3-4
              :short-name "3+4"
              :codelets
                 '((look-for-bl+ %first-threshold%
                        %second-urgency% (7 3 4)))))

(setq plus3-5 (make-instance 'pnode
              :neighbors
                 '((node-3 result+)
                   (node-5 result+)
                   (node-8 result+))
              :name 'plus3-5
              :short-name "3+5"
              :codelets
                 '((look-for-bl+ %first-threshold%
                        %second-urgency% (8 3 5)))))

(setq plus3-6 (make-instance 'pnode
              :neighbors
                 '((node-3 result+)
                   (node-6 result+)
                   (node-9 result+))
              :name 'plus3-6
              :short-name "3+6"
              :codelets
                 '((look-for-bl+ %first-threshold%
                        %second-urgency% (9 3 6)))))

(setq plus3-7 (make-instance 'pnode
              :neighbors
                 '((node-3 result+)
                   (node-7 result+)
                   (node-10 result+))
              :name 'plus3-7
              :short-name "3+7"
              :codelets
                 '((look-for-bl+ %first-threshold%
                        %second-urgency% (10 3 7)))))

(setq plus4-4 (make-instance 'pnode
              :neighbors
                 '((node-4 result+)
                   (node-8 result+))
              :name 'plus4-4
              :short-name "4+4"
              :codelets
                 '((look-for-bl+ %first-threshold%
                        %second-urgency% (8 4 4)))))
```

```
(setq plus4-5 (make-instance 'pnode
               :neighbors
                  '((node-4 result+)
                    (node-5 result+)
                    (node-9 result+))
               :name 'plus4-5
               :short-name "4+5"
               :codelets
                  '((look-for-bl+ %first-threshold%
                        %second-urgency% (9 4 5)))))
(setq plus4-6 (make-instance 'pnode
               :neighbors
                  '((node-4 result+)
                    (node-6 result+)
                    (node-10 result+))
               :name 'plus4-6
               :short-name "4+6"
               :codelets
                  '((look-for-bl+ %first-threshold%
                        %second-urgency% (10 4 6)))))
(setq plus5-5 (make-instance 'pnode
               :neighbors
                  '((node-5 result+)
                    (node-10 result+))
               :name 'plus5-5
               :short-name "5+5"
               :codelets
                  '((look-for-bl+ %first-threshold%
                        %second-urgency% (10 5 5)))))
(setq plus7-8 (make-instance 'pnode
               :neighbors
                  '((node-7 result+)
                    (node-8 result+)
                    (node-15 result+))
               :name 'plus7-8
               :short-name "7+8"
               :codelets
                  '((look-for-bl+ %first-threshold%
                        %second-urgency% (15 7 8)))))
(setq plus5-10 (make-instance 'pnode
                :neighbors
                  '((node-5 result+)
                    (node-10 result+)
                    (node-15 result+))
                :name 'plus5-10
                :short-name "5+10"
                :codelets
                  '((look-for-bl+ %first-threshold%
                        %second-urgency% (15 5 10)))))
```

```
(setq times2-2 (make-instance 'pnode
                 :neighbors
                    '((node-2 operand)
                  (node-4 resultx))
                    :name 'times2-2
                    :short-name "2x2"
                    :codelets
                       '((look-for-blx %first-threshold%
                             %second-urgency% (4 2 2)))))

(setq times2-3 (make-instance 'pnode
                 :neighbors
                    '((node-2 operand)
                      (node-3 operand)
                      (node-6 resultx))
                 :name 'times2-3
                 :short-name "2x3"
                 :codelets
                    '((look-for-blx %first-threshold%
                          %second-urgency% (6 2 3)))))

(setq times2-4 (make-instance 'pnode
                 :neighbors
                    '((node-2 operand)
                      (node-4 operand)
                      (node-8 resultx))
                 :name 'times2-4
                 :short-name "2x4"
                 :codelets
                    '((look-for-blx %first-threshold%
                          %second-urgency% (8 2 4)))))

(setq times2-5 (make-instance 'pnode
                 :neighbors
                    '((node-2 operand)
                      (node-5 operand)
                      (node-10 resultx))
                 :name 'times2-5
                 :short-name "2x5"
                 :codelets
                    '((look-for-blx %first-threshold%
                          %second-urgency% (10 2 5)))))

(setq times2-7 (make-instance 'pnode
                 :neighbors
                    '((node-2 operand)
                      (node-7 operand)
                      (node-15 similar))
                 :name 'times2-7
                 :short-name "2x7"
                 :codelets
                    '((look-for-blx %first-threshold%
                          %second-urgency% (15 2 7)))))

(setq times2-10 (make-instance 'pnode
                  :neighbors
```

```
                    '((node-2 operand)
                      (node-10 operand)
                      (node-20 resultx))
                  :name 'times2-10
                  :short-name "2x10"
                  :codelets
                     '((look-for-blx %first-threshold%
                           %second-urgency% (20 2 10)))))
(setq times2-12 (make-instance 'pnode
                  :neighbors
                     '((node-2 operand)
                       (node-12 operand)
                       (node-25 similar))
                  :name 'times2-12
                  :short-name "2x12"
                  :codelets
                     '((look-for-blx %first-threshold%
                           %second-urgency% (24 2 12)))))
(setq times2-20 (make-instance 'pnode
                  :neighbors
                     '((node-2 operand)
                       (node-20 operand)
                       (node-40 resultx))
                  :name 'times2-20
                  :short-name "2x20"
                  :codelets
                     '((look-for-blx %first-threshold%
                           %second-urgency% (40 2 20)))))
(setq times3-3 (make-instance 'pnode
                  :neighbors
                     '((node-3 operand)
                       (node-9 resultx))
                  :name 'times3-3
                  :short-name "3x3"
                  :codelets
                     '((look-for-blx %first-threshold%
                           %second-urgency% (9 3 3)))))
(setq times3-10 (make-instance 'pnode
                  :neighbors
                     '((node-3 operand)
                       (node-10 operand)
                       (node-30 resultx))
                  :name 'times3-10
                  :short-name "3x10"
                  :codelets
                     '((look-for-blx %first-threshold%
                           %second-urgency% (30 3 10)))))
(setq times3-20 (make-instance 'pnode
                  :neighbors
                     '((node-3 operand)
                       (node-20 operand)
```

```
                    (node-60 resultx))
                :name 'times3-20
                :short-name "3x20"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (60 3 20)))))

(setq times4-4 (make-instance 'pnode
                :neighbors
                   '((node-4 operand)
                     (node-16 resultx))
                :name 'times4-4
                :short-name "4x4"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (16 4 4)))))

(setq times4-10 (make-instance 'pnode
                :neighbors
                   '((node-4 operand)
                     (node-10 operand)
                     (node-40 resultx))
                :name 'times4-10
                :short-name "4x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (40 4 10)))))

(setq times4-20 (make-instance 'pnode
                :neighbors
                   '((node-4 operand)
                     (node-20 operand)
                     (node-80 resultx))
                :name 'times4-20
                :short-name "4x20"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (80 4 20)))))


(setq times5-5 (make-instance 'pnode
                :neighbors
                   '((node-5 operand)
                     (node-25 resultx))
                :name 'times5-5
                :short-name "5x5"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (25 5 5)))))

(setq times5-6 (make-instance 'pnode
                :neighbors
                   '((node-5 operand)
                     (node-6 operand)
                     (node-30 resultx))
```

```
                :name 'times5-6
                :short-name "5x6"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (30 5 6)))))

(setq times5-10 (make-instance 'pnode
                :neighbors
                   '((node-5 operand)
                     (node-10 operand)
                     (node-50 resultx))
                :name 'times5-10
                :short-name "5x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (50 5 10)))))

(setq times5-20 (make-instance 'pnode
                :neighbors
                   '((node-5 operand)
                     (node-20 operand)
                     (node-100 resultx))
                :name 'times5-20
                :short-name "5x20"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (100 5 20)))))

(setq times6-10 (make-instance 'pnode
                :neighbors
                   '((node-6 operand)
                     (node-10 operand)
                     (node-60 resultx))
                :name 'times6-10
                :short-name "6x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (60 6 10)))))


(setq times7-7 (make-instance 'pnode
                :neighbors
                   '((node-7 operand)
                     (node-50 similar))
                :name 'times7-7
                :short-name "7x7"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (49 7 7)))))

(setq times7-10 (make-instance 'pnode
                :neighbors
                   '((node-7 operand)
                     (node-10 operand)
                     (node-70 resultx))
                :name 'times7-10
```

```
                :short-name "7x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (70 7 10)))))

(setq times8-10 (make-instance 'pnode
                :neighbors
                   '((node-8 operand)
                     (node-10 operand)
                     (node-80 resultx))
                :name 'times8-10
                :short-name "8x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (80 8 10)))))


(setq times9-9 (make-instance 'pnode
                :neighbors
                   '((node-9 operand)
                     (node-81 resultx))
                :name 'times9-9
                :short-name "9x9"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (81 9 9)))))

(setq times9-10 (make-instance 'pnode
                :neighbors
                   '((node-9 operand)
                     (node-10 operand)
                     (node-90 resultx))
                :name 'times9-10
                :short-name "9x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (90 9 10)))))


(setq times10-10 (make-instance 'pnode
                :neighbors
                   '((node-10 operand)
                     (node-100 resultx))
                :name 'times10-10
                :short-name "10x10"
                :codelets
                   '((look-for-blx %first-threshold%
                         %second-urgency% (100 10 10)))))

(setq times10-15 (make-instance 'pnode
                :neighbors
                   '((node-10 operand)
                     (node-15 operand)
                     (node-150 resultx))
                :name 'times10-15
```

```
                    :short-name "10x15"
                    :codelets
                       '((look-for-blx %first-threshold%
                             %second-urgency% (150 10 15)))))

    (setq operand (make-instance 'pnode
                    :name 'operand
                    :short-name "operd"))

    (setq result+ (make-instance 'pnode
                    :name 'result+
                    :short-name "resl+"))

    (setq resultx (make-instance 'pnode
                    :name 'resultx
                    :short-name "reslx"))

    (setq similar (make-instance 'pnode
                    :name 'similar
                    :short-name "simil"))

    (setq operation (make-instance 'pnode
                    :name 'operation
                    :short-name "opion"))

    (setq instance (make-instance 'pnode
                    :name 'instance
                    :short-name "inst"))

    (setq plus (make-instance 'pnode
                    :name 'plus
                    :short-name "plus"))

    (setq minus (make-instance 'pnode
                    :name 'minus
                    :short-name "diff"))

    (setq times (make-instance 'pnode
                    :name 'times
                    :short-name "prod"))

    )

(init-pnet)

(setq *pnet* (list node-1 node-2 node-3 node-4 node-5 node-6 node-7 node-8
                   node-9 node-10 node-12 node-15 node-16 node-20 node-25
                   node-30 node-40 node-50 node-60 node-70 node-80
                   node-81 node-90 node-100 node-150
                   plus1-1 plus1-2 plus1-3 plus1-4 plus1-5 plus1-6 plus1-7
                   plus1-8 plus1-9 plus2-2 plus2-3 plus2-4 plus2-5 plus2-6
                   plus2-7 plus2-8 plus3-3 plus3-4 plus3-5 plus3-6 plus3-7
                   plus4-4 plus4-5 plus4-6 plus5-5 plus5-10 plus7-8
                   times2-2 times2-3 times2-4 times2-5 times2-7
                   times2-10 times2-12 times2-20
                   times3-3 times3-10 times3-20 times4-4 times4-10
```

```
                times4-20 times5-5 times5-6 times5-10 times5-20
                times6-10  times7-7 times7-10 times8-10 times9-9
                times9-10 times10-10 times10-15
                node-mul0 node-multiply node-add node-subtract
                operand result+ resultx similar operation instance))


(defun print-pnet-all ()
; Prints all the nodes and activations in the pnet.
   (loop for pnode in *pnet* do
       (format t "(~a ~a)"
               (send pnode :name)
               (send pnode :activation))))

(defun print-pnet (threshold)
; Prints the nodes and activations that are > threshold.
   (loop for pnode in *pnet* do
     (let (act)
       (if (> (setq act (send pnode :activation)) threshold)
        then (format t "(~a ~a)"
              (send pnode :name)
              (send pnode :activation))))))

(compile-flavor-methods pnode)
```

```
(declare (macros t))

;ACTIVATION-DECAY METHOD
(defmethod (pnode :activation-decay) ()
; Computes and returns the activation decay of the pnode.
   (times (my :activation-decay-factor) (my :activation)))


;ACTIVATION-DECAY-FACTOR METHOD
(defmethod (pnode :activation-decay-factor) ()
; The decay is linked to the type of instance in the cyto
  (let ((s (caar (sortcar (my :instances) ()))))
   (cond ((equal s "1t") %first-decay-rate%)
         ((equal s "2b") %second-decay-rate%)
         ((equal s "3dt") %third-decay-rate%)
         ((equal s "5g") %fifth-decay-rate%)
         ((equal s "6g") %sixth-decay-rate%)
         (t  %fourth-decay-rate%))))

;ADD-ACTIVATION METHOD
;This method adds the given amount of activation to the pnode
; (if the amount to be added is above the %min-activation-to-be-added%
; threshold).
;The amount can be negative. In that case, the activation is
;decreased, but can never be negative.
(defmethod (pnode :add-activation) (act)
    (if (> (abs act) %min-activation-to-be-added%)
     then (setq activation (max 0 (plus activation act)))))

;ADD-TEMP-ACTIVATION-HOLDER METHOD
(defmethod (pnode :add-temp-activation-holder) (act)
    (send self :set-temp-activation-holder
               (plus (my :temp-activation-holder) act)))

;CODELET-URGENCY METHOD
(defmethod (pnode :codelet-urgency) (base-urgency)
; Calculates what the urgency of this codelet should be, given the
; base urgency of the codelet.
   base-urgency)

;HOTTER-NEIGHBOR-ACTIVATION METHOD
(defmethod (pnode :hotter-neighbor-activation) ()
;this method returns the activation of the hottest neighbor which
;is not a brick
    (let ((maxi 0) act)
         (loop for link in (my :neighbors) do
;Here link is of the form (node link-type)
  (setq node (car link))
  (cond
       ((eq (cadr link)  similar) nil)
       (t
          (setq act (send (car link) :activation))
          (if (> act maxi) then (setq maxi act)))))
        maxi))

;INITIALIZE-CODELET FUNCTION
```

```
(defun initialize-codelet ()
; Loops through the nodes in the pnet, examining codelets of
; all the nodes. If a codelets variable contains a codelet with a
; corresponding threshold nequal to %first-threshold% this threshold
; is replaced by %first-threshold%.
  (let ((x 0))
   (loop for pnode in *pnet* do
      (loop for codelet in (send pnode :codelets) do
            (let ((codelet-call (car codelet))
                  (threshold (cadr codelet))
                  (base-urgency (caddr codelet))
                  (arguments (cadddr codelet)))
            (if (nequal (eval threshold)  (eval %first-threshold%))
             then
                  (send pnode :modify-threshold codelet-call
                        '%first-threshold%)))))))

;INITIALIZE-PNET FUNCTION
(defun initialize-pnet ()
  (loop for pnode in *pnet* do
     (send pnode :set-activation %initial-activation%)
     (send pnode :set-temp-activation-holder 0.0)
     (send pnode :set-spreadable-activation 0.0)
     (send pnode :set-instances nil))
  (initialize-codelet))

;INITIALIZE-PNET-2 FUNCTION
; Now evaluate all the atoms in the :neighbors
(defun initialize-pnet-2 ()
  (loop for pnode in *pnet* do
    (send pnode :set-neighbors
          (loop for pair
                in (send pnode :neighbors)
                collect (list (eval (car pair)) (eval (cadr pair)))))))

;LINK-LENGTH METHOD
(defmethod (pnode :link-length) (&optional (k 0.1))
; Computes and returns the length of the links associated with this pnode.
; The length of the link is inversely proportional to the activation.
; When activation is 0, then the link-length is %length%, when activation
; is infinite, it is 1.  The minimum link-length is 1 because we
; are going to calculate the amount of "radiated energy" passed
; from one node to its neighbors as an inverse square law, i.e.
; the activation passed will be the decayed activation divided
; by the square of the link-length between the nodes.  The function
; used is f[x] = 1 + 1/[kx + 1/%length%] where %k% is a constant which
; depends on the link.
    (plus 1 (quotient (float 1) (plus (times %k% (my :activation))
                                      (quotient (float 1) %length%)))))

;MODIFY-THRESHOLD METHOD
(defmethod (pnode :modify-threshold) (codelet-call &optional (new nil))
;Replaces the threshold of a given codelet in the codelets variable
;of a pnode by a new threshold which is a function.
;This new threshold starts with a value equal to %upper-threshold%
;and diminishes with time.
```

```
;Every basic iteration (x counter) the value is decreased by 1.
 (declare (special *iteration*))
 (let ((res nil) (function nil))
      (if (null new)
      (setq new (list 'max %first-threshold% (list 'add *iteration*
                                                   '(minus *iteration*)
                                                   %upper-threshold%))))
      (loop for codelet in codelets do
           (setq function (car codelet))
           (if (equal codelet-call function)
               then
            (setq codelet (cons function (cons new (cddr codelet)))))
           (setq res (cons codelet res)))
      (setq codelets res)))

;POPULATE-CODERACK FUNCTION
(defun populate-coderack ()
; Loops through the nodes in the pnet, examining activations of
; all the nodes. Whichever nodes exceed the threshold activation get
; to post their codelet(s), whose urgency is a function of the activation
; of the node. Note: the value of the pnode slot :CODELETS is a list
; of triples of the form (function-call threshold-activation base-urgency)
; where the threshold-activation value is the one used in determining if
; base-urgency.
   (loop for pnode in *pnet* do
         (loop for codelet in (send pnode :codelets) do
               (let ((codelet-call (car codelet))
                     (threshold (cadr codelet))
                     (base-urgency (caddr codelet))
                     (arguments (cadddr codelet)))
               (if (>= (send pnode :activation) (eval threshold))
                then (if %verbose%
                         then (format t "About to post codelet ~a ~a~&"
                                      codelet-call arguments))
                     (cr-hang *coderack*
                       (append (list codelet-call) arguments)
                       (send pnode :codelet-urgency (eval base-urgency)))
                       (send pnode :modify-threshold codelet-call
                             ))))))

;PRINT METHOD
(defmethod (pnode :print) ()
  (format t "I am a pnode.~&")
  (format t "name: ~a~&" (my :name))
  (format t "instances: ~a~&" (my :instances))
  (format t "activation: ~a~&" (my :activation))
  (format t "spreadable-activation: ~a~&" (my :spreadable-activation))
  (format t "temp-activation-holder: ~a~&" (my :temp-activation-holder))
  (format t "neighbors: ~a~&"
            (my :neighbors)))

;SET-UP-ACTIVATIONS FUNCTION
(defun set-up-activations (list-of-pnodes-and-activations)
; Sets up activations as in the list given in the argument. This is a list
```

```
; of the form (node1 act1 node2 act2 . . .)
    (loop until (null list-of-pnodes-and-activations) do
        (send (eval (car list-of-pnodes-and-activations))
              :set-activation (cadr list-of-pnodes-and-activations))
        (setq list-of-pnodes-and-activations
              (cddr list-of-pnodes-and-activations))))

;SPREAD-ACTIVATION METHOD
(defmethod (pnode :spread-activation) ()
; Spreads activation to all nodes the given node is linked to.
; The amount of activation spread along each link is
;                :spreadable-activation / (:link-length ^ 2)
; where :spreadable-activation is d * (activation of node doing the
; spreading).
; where d is the decay factor of the node doing the spreading.

   (loop for link in (my :neighbors) do
       ; here, link is of the form (node link-type)
       (let (linked-node link-type link-length)
          (setq linked-node (car link))
          (setq link-type (cadr link))
          (setq link-length (send link-type :link-length))
          ; give linked-node new activation (store in temporary holder)
          (send linked-node :add-temp-activation-holder
                            (quotient (my :spreadable-activation)
                                      (sqrt link-length))))))

;SPREAD-ACTIVATION-IN-PNET METHOD
(defun spread-activation-in-pnet ()
; Causes each node to decay and then
; Spreads activation throughout the pnet. All the
; nodes spread activation in "parallel" and the spread
; of activation is "instantaneous". This is simulated as
; follows: For each node, the amount of spreadable activation is computed
; and put in the :spreadable-activation instance variable.
; This is also the amount that the node will have "decayed" during this
; cycle, and it will be subtracted later from the node's activation.
; Next each node in turn spreads activation to other nodes, the spread
; activation being put in the :temp-activation-holder variable in each node.
; When the entire pnet has been gone over, each node's activation is
; then set to its old activation plus :temp-activation-holder minus
; :spreadable-activation. Thus the new activation of a node is
;          new-activation (node) = old-activation (node) - decay +
;                                  activation spread from other nodes.
;

   ; Compute and store :spreadable-activation for each node.
   (loop for pnode in *pnet* do
       (send pnode :set-spreadable-activation
                   (send pnode :activation-decay)))

   ; Spread activation from each node to its neighbors
   (loop for pnode in *pnet* do
       (send pnode :spread-activation))

   ; Update activation for each node.
```

```
   (loop for pnode in *pnet* do
       (send pnode :update-activation)))

;SUBTRACT-ACTIVATION METHOD
(defmethod (pnode :subtract-activation) (act)
    (if (> act %min-activation-to-be-added%)
     then (setq activation (difference activation act))))

;SUPPRESS-INSTANCES METHOD
(defmethod (pnode :suppress-instances) (l)
; Suppress a given cyto-node from the list of instances.
     (setq res nil)
     (do ((x instances (cdr x)))
         ((null x) (setq instances res))
         (cond
             ((eq (cadar x) l) nil)
             (t (setq res (cons (car x) res))))))

;UPDATE-ACTIVATION METHOD
(defmethod (pnode :update-activation) ()
; Sets the activation of the node to old-activation - decay
; (= :spreadable-activation) + :temp-activation-holder, and then
; sets :spreadable-activation and :temp-activation-holder to zero.
;If the activation to be added is not high enough (less than
;%min-activation-to-be-added% ),nothing is added (if the node is
;not a numerical one)
(let ((activation-to-add temp-activation-holder) new-activation)
   (if
      (and (null value)
           (< activation-to-add  %min-activation-to-be-added%))
       (setq activation-to-add 0))
    (setq activation-to-add (min %max-activation-to-be-transmitted%
                                 activation-to-add))
    (setq new-activation (plus (difference (my :activation)
         (my :spreadable-activation)) activation-to-add))
    (send self :set-activation new-activation)
    (send self :set-spreadable-activation 0.0)
    (send self :set-temp-activation-holder 0.0)))

;UPDATE-INSTANCES METHOD
(defmethod (pnode :update-instances) (l)
; Makes it possible to link a cyto-node to a pnode by updating instances
; Instances is a list of pairs. Each pair specify the type of the
; cyto-node (1t 2b 3dt 4bl 5g) and its name.
     (setq instances (cons l instances)))

(compile-flavor-methods pnode)
```

```
; Note: some of the stuff in here was done in a kludgey way so it would
; work fast for my purposes. If anyone else is using these routines, they
; might want to make things more general.

(declare (special
          max-activation
          text-size
          text-offset
          origin-x origin-y
          wheight
          wwidth
          num-of-chars
          region-side
          diminished-region-side
          new-boxsize
          new-box-coordinates
          boxsizes
         )
)


(defun init-pnet-graphics (node-list)
; Set up global variables
    (open-window)
    (clear-window)
    (setq max-activation 256) ; maximum activation a node can have
    (setq text-size 5) ; pixel size of text (approximate)
    (setq wheight (window-height))
    (setq wwidth  (window-width))
    (setq origin-x 0) ; x-coordinate of origin of display
    (setq origin-y 0) ; y-coordinate of origin of display
    (setq num-of-chars 3) ; number of characters allowed in title
    (setup-regions node-list)  ; set up the regions in which the nodes will
                               ; be displayed
    (make-boxsizes)  ; sets up a global vector of the boxsize for each
                     ; possible activation
    (loop for node in node-list do (send node :set-old-boxsize 0)))


(defun display-pnet (node-list)
    (outline-regions node-list)
    (update-pnet-display node-list))


(defun setup-regions (node-list)
; Sets up the display for the pnodes in the given list.
; This function first calculates the area of the graphics window,
; WINDOW-AREA. Then it calculates the area each square pnode
; region will get (REGION-AREA = WINDOW-AREA / NUM-OF-NODES),
; and proceeds to assign coordinates for the upper-left-hand
; corners of each pnode region.

    (let ((num-of-nodes (length node-list))
          (dummy-list node-list)
          window-area region-area
          num-of-regions-in-width num-of-regions-in-height)
```

```
    (setq window-area (* wwidth wheight))
    (setq region-area (/ window-area num-of-nodes))
    (setq region-side  (fix (sqrt region-area)))


; Here I assume height > width, since I'm going to use windows where
; that's true. (I.e., this is a kludge.)
    (setq num-of-regions-in-height (/ wheight region-side))
    (setq num-of-regions-in-width (1+ (/ wwidth region-side)))
; Make sure there are enough regions
   (if (< (* num-of-regions-in-height num-of-regions-in-width)
          num-of-nodes)
    then  ; this is a kludge, but it works
        (if (or (= num-of-regions-in-height
                   (1+ num-of-regions-in-width))
                (= num-of-regions-in-height
                   num-of-regions-in-width))
         then (incf num-of-regions-in-height)
         else
             (setq num-of-regions-in-height
                   (1- num-of-regions-in-height))
             (setq num-of-regions-in-width
                   (1+ num-of-regions-in-width))))
; Make sure all the regions will fit on the screen
    (if (< wwidth (* num-of-regions-in-width region-side))
     then (setq region-side (/ wwidth num-of-regions-in-width)))
; the diminished region size is the max size box for
; displaying a node
    (setq diminished-region-side (fix (times region-side .70)))
    ; offset for drawing in text
    (setq text-offset (fix (times diminished-region-side .02)))
    (loop for i from 0 to (1- num-of-regions-in-width) do
       (loop for j from 0 to (1- num-of-regions-in-height) do
          (let ((node (car dummy-list)))
             (send node :set-x-region (+ origin-x (* i region-side)))
             (send node :set-y-region (+ origin-y (* j region-side)))

             (send node :set-x-act (+ diminished-region-side
                                      (minus text-size)
                                      (send node :x-region)))
             (send node :set-y-act (+ (* 3 text-size)
                                      (send node :y-region)))
             (setq dummy-list (cdr dummy-list)))

        until (null dummy-list)) until (null dummy-list))))


(defun make-boxsizes ()
; Sets up a global vector containing the boxsizes, indexed by activation.
    (setq boxsizes (new-vector (1+ max-activation)))
    (loop for activation from 0 to max-activation do
       (vset boxsizes activation
            (round-off (times (float diminished-region-side)
                              (quotient (float activation)
                                        (float max-activation)))))))
```

```
(defun outline-regions (node-list)
; Outlines the regions for displaying the pnet
    (loop for node in node-list do
       (let* ((box-x (send node :x-region))
              (box-y (send node :y-region)))
         (draw-unfilled-rect box-x box-y
                             (+ box-x region-side)
                             (+ box-y region-side))
          (draw-text (+ box-x text-offset)
                     (- (+ box-y region-side) text-offset)
                     (send node :short-name)))))


(defmethod (pnode :new-boxsize) ()
; Computes the box size for the node (a function of its current activation).
; The function for box size of a node is
;  region-side * (activation / max-activation)
; If activation > max-activation, then max-activation is used.

      (cond ((> activation max-activation)
             diminished-region-side)  ; max size
            ((< activation 0)
             0)
            (t (vref boxsizes (fix activation)))))


(defmethod (pnode :new-box-coordinates) ()
; Returns a dotted pair of the x and y coordinates of the upper-left-hand
; corner of the box that is to be drawn.
  (cons (+ x-region
           (/ (- region-side (send self :new-boxsize)) 2))
        (+ y-region
           (/ (- region-side (send self :new-boxsize)) 2))))


(defmethod (pnode :draw-box) ()
; Draws the box corresponding to the current pnode

 (let ((old-boxsize old-boxsize)
       (new-boxsize (send self :new-boxsize))
       (new-x-box (car (send self :new-box-coordinates)))
       (new-y-box (cdr (send self :new-box-coordinates))))
 ; If old box size > new-boxsize then shrink old box
       (if (> old-boxsize new-boxsize)
        then (if (< (- old-boxsize new-boxsize) 10)
              then (erasebox old-boxsize x-box y-box)
                   (drawbox new-boxsize new-x-box new-y-box)
              else (shrink-box old-boxsize new-boxsize
                               x-box y-box
                               new-x-box new-y-box))
        else  ; expand old box
          (if (> activation 0)
           then (drawbox new-boxsize new-x-box new-y-box)))
```

```
; erase old-activation and display new-activation numbers
        (let ((x-coord x-act)
              (y-coord y-act))
         (erase-rect x-coord
                     (- y-coord 10)
                     (+ x-coord 30)
                     y-coord)


         (draw-number x-coord y-coord (fix activation)))

; Save size of box and coordinates
         (send self :set-old-boxsize (send self :new-boxsize))
         (send self :set-x-box new-x-box)
         (send self :set-y-box new-y-box)))


(defun update-pnet-display (node-list)
; Displays boxes corresponding to the nodes in the node-list
      (loop for node in node-list do
          (if (> (send node :activation) 0)
           then (send node :draw-box))))


(defun shrink-box (oldboxsize newboxsize oldx oldy newx newy)
; Shrinks box from oldboxsize to newboxsize.
   (let ((var1 (+ oldx oldboxsize))
         (var2 (+ oldy oldboxsize))
         (var3 (+ newx newboxsize))
         (var4 (+ newy newboxsize)))
     (erase-rect oldx oldy var1 newy)
     (erase-rect var3 newy var1 var2)
     (erase-rect oldx var4 var1 var2)
     (erase-rect oldx oldy newx var4)))

(defun drawbox (boxsize x y)
; Draws a filled box with upper-left-hand corner (x y) and side boxsize.
   (draw-rect x y (+ x boxsize) (+ y boxsize)))


(defun erasebox (boxsize x y)
; Erases a box with upper-left-hand corner (x y) and side boxsize.
   (erase-rect x y (+ x boxsize) (+ y boxsize)))


(defun round-off (realnum)
; Rounds off a real number (to an integer).
   (fix (plus realnum 0.5)))
```

```
; CYTOPLASM FLAVOR
; The cytoplasm is represented as a flavor with instance variable
; nodes ,current-target and context
(defflavor cytoplasm
           (target
            brick1
            brick2
            brick3
            brick4
            brick5
            nodes
            current-target
            context)
           ()
; Cytoplasm       - target and bricks are the parameters of the game.
;                 - nodes are the nodes which constitute the cyto.
;                   They are flavors as well (cyto-node).
;                 - current-target is a flavor which the target of current
;                       interest at a given time.
;                 - context is a flavor with the current entities of interest
;                                                 (very short term storage)
           :settable-instance-variables
           :inittable-instance-variables
           :gettable-instance-variables)

; CYTO-NODE FLAVOR
(defflavor cyto-node
           (activation
            neighbors
            name
            type
            value
            success
            status
            level
            listed
            plinks)
           ()
; Cyto-node       - activation is the level of activation
;                 - neighbors are the neighbors and the corresponding type of link
;                 - name is the full name of the node.It is also the name of
;                     the variable which contains the node address.
;                 - type is the type of the node(1t 2b 3dt 4bl 5g).
;                 - value is the possible value associated with the node.
;                 - success tells us if a target is hit or not (1 0).
;                 - status tells us if the node is still free or is already or is l
;                 - level is the level of the node in the tree. By definition
;                   the level of the bricks is 1 and of the target 99.
;                   For blocks the level is increasing, for derived targets
;                   it is decreasing.
;                                 implied in some operation (free linked).
;                 - listed is used in listing the solution . Its value is yes
;                     for a given operation when it has been listed in the
;                                 final result.
;                 - plinks are the links with the pnodes (list of pnodes),
;                               but  the address is not compiled.
```

```
            :settable-instance-variables
            :inittable-instance-variables
            :gettable-instance-variables)

; CYTO-CURRENT-TARGET
(defflavor cyto-current-target
           (interest
            name)
            ()
; Cyto-current-target    - interest is the current interest in the target
;                        - name is the name of the node which represents the
;                                                        current target
;
            :settable-instance-variables
            :inittable-instance-variables
            :gettable-instance-variables)

; CYTO-CONTEXT
(defflavor cyto-context
           (type
            name
            value
            interest
            location)
           ()
; Cyto-context-node      - type is the type of the node if relevant
;                        - name is the name of the node which represents the
;                                                        current target
;                        - value is the possible value associated with the node
;                        - interest is the current interest in the node
;                        - location is Pnet or Cytoplasm
;
            :settable-instance-variables
            :inittable-instance-variables
            :gettable-instance-variables)

;CYTO-BRICK-BLOCK-NODES METHOD
(defmethod (cytoplasm :cyto-brick-block-nodes) ()
;Sends the list of cyto-bricks and cyto-blocks
;in the cytoplasm nodes variable
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq type (send (car x) :type))
            (cond
                 ((equal "1t" type) nil)
                 ((equal "3dt" type) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (cons (car x) res)))))))

;FIND-NEW-TARGET METHOD
(defmethod (cytoplasm :find-new-target) ()
;Sends the free target from the nodes variable of the cytoplasm.
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq type (send (car x) :type))
            (setq status (send (car x) :status))
```

```
            (cond
                 ((equal "linked" status) nil)
                 ((equal "2b" type) nil)
                 ((equal "4bl" type) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (car x) ))))))

;FREE-BLOCKS METHOD
(defmethod (cytoplasm :free-blocks) ()
;Sends the list of blocks in the cytoplasm nodes variable which are
;still free.
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq type (send (car x) :type))
            (setq status (send (car x) :status))
            (cond
                 ((equal "linked" status) nil)
                 ((equal "1t" type) nil)
                 ((equal "3dt" type) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (cons (car x) res)))))))

;FREE-CYTO-NODES METHOD
(defmethod (cytoplasm :free-cyto-nodes) ()
;Sends the list of cyto-nodes in the cytoplasm nodes variable
;which are still free.
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq status (send (car x) :status))
            (setq type (send (car x) :type))
            (cond
                 ((equal "linked" status) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (cons (car x) res)))))))

;FREE-SECONDARY-CYTO-NODES METHOD
(defmethod (cytoplasm :free-secondary-cyto-nodes) ()
;Sends the list of secondary-cyto-nodes in the cytoplasm nodes variable
;which are still free. Bricks,target and operations are thus excluded.
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq status (send (car x) :status))
            (setq type (send (car x) :type))
            (cond
                 ((equal "linked" status) nil)
                 ((equal "1t" type) nil)
                 ((equal "2b" type) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (cons (car x) res)))))))

;INIT-CYTOPLASM FUNCTION
;Initialize the cytoplasm
(defun init-cytoplasm (target brick1 brick2 brick3 brick4 brick5)
```

```
    (setq *temperature* 100)
    (setq *context* (make-instance 'cyto-context))
    (setq *current-target* (make-instance 'cyto-current-target))
    (setq *cytoplasm* (make-instance 'cytoplasm
                              :target target
                              :brick1 brick1
                              :brick2 brick2
                              :brick3 brick3
                              :brick4 brick4
                              :brick5 brick5
                              :current-target *current-target*
                              :context *context*)))

;LOWER-DTARGET-NEIGHBOR METHOD
(defmethod (cyto-node :lower-dtarget-neighbor) ()
;This method gives the lower neighbor which has the type "3dt".
(let ((res nil) (lev 1000))
    (loop for pair in neighbors do
          (setq lv (send (car pair) :level))
          (cond
               ((and (equal "3dt" (send (car pair) :type))
                     (< lv lev)) (setq lev lv)
                                 (setq res (car pair)))
               (t nil)))
    res))

;LOWER-NEIGHBOR METHOD
(defmethod (cyto-node :lower-neighbor) ()
;This method gives the lower neighbor if the node is not
;an operation node
(let ((res nil) (lev 1000))
    (cond
         ((equal "5g" type) nil)
         (t
          (loop for pair in neighbors do
           (setq lv (send (car pair) :level))
           (cond
                ((< lv lev) (setq lev lv)
                            (setq res (car pair)))
                (t nil)))))
    res))

;BLOCK-NEIGHBOR METHOD
(defmethod (cyto-node :block-neighbor) ()
;The method determines for a derived target, its block neighbor
;In fact, the block neighbor of its neighbor. Used to avoid to
;repeat associations of a target with the same bock.
(let (node l (res nil))
  (cond
      ((nequal "3dt" type) nil)
      ((nequal "free" status) nil)
      (t
;l will contain the neighbors of the operation node linked to
;cyto-node
       (setq l (send (send self :upper-neighbor) :neighbors))
;The node of l which is a block or a brick is put in res
```

```
        (loop for pair in l do
              (setq node (car pair))
              (cond
                  ((equal "2b" (send node :type)) (setq res node))
                  ((equal "4bl" (send node :type)) (setq res node))
                  (t nil)))))
  res))

;REPLACE FUNCTION
(defun replace-function (l l1 l2)
;This functions replace in l which is a list of couples, all the couples
;which start by l1 by a corresponding couple starting with l2 .
(let ((res nil) pair)
    (do ((x l (cdr x)))
        ((null x) res)
        (cond
             ((eq (caar x) l1) (setq pair (cons l2 (cdar x))))
             (t (setq pair (car x))))
        (setq res (cons pair res)))))

;REPLACE-NEIGHBORS METHOD
(defmethod (cyto-node :replace-neighbors) (l1 l2)
;This method replaces in the list of neighbors of a cyto-node
;the node l1 by the node l2. It uses the function
;replace in mylist l1 by l2 (replace mylist l1 l2).
      (setq neighbors  (replace-function neighbors l1 l2)))

;SECONDARY-CYTO-NODES METHOD
(defmethod (cytoplasm :secondary-cyto-nodes) ()
;Sends the list of secondary-cyto-nodes (blocks and dtargets)
;in the cytoplasm nodes variable
       (let ((res nil))
       (do  ((x nodes (cdr x)))
            ((null x) res)
            (setq type (send (car x) :type))
            (cond
                 ((equal "1t" type) nil)
                 ((equal "2b" type) nil)
                 ((equal "5g" type) nil)
                 (t (setq res (cons (car x) res)))))))

;SUPPRESS-NEIGHBORS METHOD
(defmethod (cyto-node :suppress-neighbors) (l1)
;This methods suppresses the node l1 (in fact the couple
;which starts with l1) in the list of neighbors
;of a cyto-node.
(let ((res nil))
    (do ((x neighbors (cdr x)))
        ((null x) (setq neighbors res))
        (cond
             ((eq (caar x) l1) nil)
             (t (setq res (cons (car x) res)))))))

;SUPPRESS-NODE METHOD
(defmethod (cytoplasm :suppress-node) (l)
;Suppress a given node in the nodes variable of the cytoplasm
```

```
(let ((res nil))
      (do ((x nodes (cdr x)))
          ((null x) (setq nodes res))
          (cond
                ((eq (car x) l) nil)
                (t (setq res (cons (car x) res)))))))

;UPDATE-CONTEXT FUNCTION
(defun update-context (type name value interest location)
;Updates the variables of the cyto-context node
    (send *context* :set-type type)
    (send *context* :set-name name)
    (send *context* :set-value value)
    (send *context* :set-interest interest)
    (send *context* :set-location location))

;UPDATE-CURRENT-TARGET FUNCTION
(defun update-current-target (name interest)
;Updates the variables of the cyto-current-target node
    (send *current-target* :set-name name)
    (send *current-target* :set-interest interest))

;UPDATE-NEIGHBORS METHOD
(defmethod (cyto-node :update-neighbors) (l)
;Appends a list l of neighbors to the current list
       (setq neighbors (append l Neighbors)))

;UPDATE-PLINKS METHOD
(defmethod (cyto-node :update-plinks) (l)
;This method makes it possible to add one pnode on the list of plinks.
    (setq plinks (cons l plinks)))


;UPPER-NEIGHBOR METHOD
(defmethod (cyto-node :upper-neighbor) ()
;This method gives the upper neighbor
(let ((res nil) (lev level))
     (loop for pair in neighbors do
           (setq lv (send (car pair) :level))
           (cond
                ((> lv lev) (setq lev lv)
                            (setq res (car pair)))
                (t nil)))
     res))
```

```
;ACTIVATE CODELET
(defun activate (activation pnode repump-yes-or-no)
;Activates a given pnode to a given level
;If the pnode is linked to a target, the add, subtract and multiply
;pnodes are activated as a function of the value of pnode
(let (val val- valx)
     (send (eval pnode) :add-activation activation)
     (cond
          (repump-yes-or-no
              (setq val (send pnode :value))
              (setq val- (plus 5 (times 50 (expt 0.9 val))))
              (setq valx (times 4 (sqrt val)))
              (send node-add :set-activation (diff 60 val- valx))
              (send node-subtract :set-activation val-)
              (send node-multiply :set-activation valx)
              (repump))
          (t nil))))

;CHECK-TEMPERATURE FUNCTION
(defun check-temperature function ()
;This functions checks if the temperature of the cytoplasm is not too
;high. If it is the case, a node to be killed is randomly choosen
;according to the misfortunes. A "kill-node" codelet is loaded.
(let ((rvictim 0) (victim nil))
    (cond
        ((< %temperature-threshold%  (temperature))
;         (terpri) (terpri) (format t " TEMP  ~a" (temperature))
         (setq reserve (send *cytoplasm* :secondary-cyto-nodes))
;the probability of being killed is a function of the activation
;of the node (proport. to 300 - activation)
         (setq weights (mapcar 'diff300 (find-activations reserve)))
         (setq rvictim (randlist weights))
        (cond
         (rvictim
         (setq victim (nth rvictim reserve))
         (cr-hang *coderack* (list 'kill-node victim) %first-urgency%)))))))

;COLLECT-MISFORTUNE FUNCTION
(defun collect-misfortune ()
;This function creates a list of the misfortunes of the secondary-
;cyto-nodes.
(let ((list nil) (cyto-nodes nil) (hap nil))
    (setq current-target (send *current-target* :name))
    (setq cyto-nodes (send *cytoplasm* :secondary-cyto-nodes))
    (loop for node in cyto-nodes do
          (setq hap (misfortune (send node :activation)
                                (send node :level)
                                (send node :status)))
          (setq list (cons hap list)))
    (reverse list)))

;COMPARE FUNCTION
(defun compare (element list)
;returns t if it exists in the list one number similar
;to element
(do
```

```
    ((x list (cdr x)))
    ((null x) nil)
    (if (< (sim element (car x)) 4) (return t))))

COMPARE-B-TO-T CODELET
(defun compare-b-to-t (cyto-block)
Compares a given block to the current target.
If equality then it loads a replace-target codelet and frees the block
if it is linked.
If similar (see sim function) then it loads a decomp+ codelet.
If digits in common (see digits-in-common function) then it loads a decompi
codelet.
If multiple (see multiple function) then it loads a decompx codelet.
(let (cyto-current-target current-target block stat target)
(setq target (send cyto-target :value))
(setq cyto-current-target (send *current-target* :name))
(setq current-target (send (eval cyto-current-target) :value))
(setq block (send cyto-block :value))
(setq stat (send cyto-block :status))
 (cond
  ((or (= 0 (sim block current-target)) (= 0 (sim block target)))
   (cond
        ((is-linked-to-target cyto-block) nil)
        (t
          (if (equal "linked" stat)
           (kill-block (send (send cyto-block :upper-neighbor)
                             :upper-neighbor)))
          (cr-hang *coderack* (list 'replace-target
           cyto-block cyto-current-target) %upper-urgency%))))
  ((nequal "free" stat) nil)
  (t
     (cond
        ((= 1 (sim block current-target))
(cr-hang *coderack* (list 'decomp+ cyto-block cyto-current-target)
                                            %first-urgency%))
        ((= 2 (sim block current-target))
(cr-hang *coderack* (list 'decomp+ cyto-block cyto-current-target)
                                            %second-urgency%))
        ((= 3 (sim block current-target))
(cr-hang *coderack* (list 'decomp+ cyto-block Cyto-current-target)
                                            %fourth-urgency%)))))
     (cond
        ((null (digits-in-common block current-target)) nil)
        (t
(cr-hang *coderack* (list 'decompi cyto-block cyto-current-target
(list 'quote (digits-in-common block current-target))) %second-urgency%)))))

;CONST-BL+ CODELET
(defun const-bl+ (cyto-block1 cyto-block2 sum)
;Creates a new block in the cytoplasm.
;Checks if the nodes are still free.
;Changes their status to "linked".
;Computes how to combine the blocks to get sum.
;Creates a new block in the cytoplasm.
;Creates a node x in the cytoplasm. Increments count to distinguish
;possible different versions of versions and targets.
```

```
(let (stblock1 stblock2 block1 block2 lev1 lev2 lev (activation 200)
      cyto-res cyto-op1 cyto-op2 cyto-op-node cyto-block-sum)
  (setq stblock1 (send cyto-block1 :status))
  (setq stblock2 (send cyto-block2 :status))
  (setq block1 (send cyto-block1 :value))
  (setq block2 (send cyto-block2 :value))
  (setq lev1 (send cyto-block1 :level))
  (setq lev2 (send cyto-block2 :level))
  (setq lev (max lev1 lev2))
;Checks if the blocks are still free
  (cond
      ((or (nequal "free" stblock1) (nequal "free" stblock2)) nil)
;Changes their status to "linked"
      (t
        (send cyto-block1 :set-status "linked")
        (send cyto-block2 :set-status "linked")
;Creates a new block in the cytoplasm
        (setq cyto-block-sum (concat 'cyto-block sum '-v *name-counter*))
        (if (= 0 (mod sum 5)) (setq activation 250))
        (if (= 0 (mod sum 10)) (setq activation 300))
        (create-cyto-node activation cyto-block-sum sum "free"
            (+ lev 2) "4bl" 1)
;Creates a node x in the cytoplasm
        (cond
          ((equal sum (+ block1 block2))
           (setq cyto-res (eval cyto-block-sum))
           (setq cyto-op1 cyto-block1)
           (setq cyto-op2 cyto-block2))
          ((equal sum (- block1 block2))
           (setq cyto-res cyto-block1)
           (setq cyto-op1 cyto-block2)
           (setq cyto-op2 (eval cyto-block-sum)))
          ((equal sum (- block2 block1))
           (setq cyto-res cyto-block2)
           (setq cyto-op1 cyto-block1)
           (setq cyto-op2 (eval cyto-block-sum))))
          (setq cyto-op-node (concat 'plus
               (min (send cyto-op1 :value) (send cyto-op2 :value)) '-
               (max (send cyto-op1 :value) (send cyto-op2 :value)) '-v *name-coun
          (create-op-node cyto-op-node cyto-res cyto-op1 cyto-op2
              50 (+ lev 1))
;Increments count.
          (setq *name-counter* (+ 1 *name-counter*))))))

;CONST-BLX CODELET
(defun const-blx (cyto-block1 cyto-block2 prod)
;Creates a new block in the cytoplasm.
;Checks if the nodes are still free.
;Changes their status to "linked".
;Creates a new block in the cytoplasm.
;Creates a node x in the cytoplasm. Increments count to distinguish
;possible different versions of versions and targets.
(let (stblock1 stblock2 block1 block2 lev1 lev2 lev cyto-block-prod
      (activation 200) cyto-op-node)
  (setq stblock1 (send cyto-block1 :status))
  (setq stblock2 (send cyto-block2 :status))
```

```
    (setq block1 (send cyto-block1 :value))
    (setq block2 (send cyto-block2 :value))
    (setq lev1 (send cyto-block1 :level))
    (setq lev2 (send cyto-block2 :level))
    (setq lev (max lev1 lev2))
;Checks if the blocks are still free
    (cond
        ((or (nequal "free" stblock1) (nequal "free" stblock2)) nil)
;Changes their status to "linked"
        (t
          (send cyto-block1 :set-status "linked")
          (send cyto-block2 :set-status "linked")
;Creates a new block in the cytoplasm
          (setq cyto-block-prod (concat 'cyto-block prod '-v *name-counter*))
          (if (= 0 (mod prod 5)) (setq activation 250))
          (if (= 0 (mod prod 10)) (setq activation 300))
          (create-cyto-node activation cyto-block-prod prod "free"
              (+ lev 2) "4bl" 1)
;Creates a node x in the cytoplasm
          (setq cyto-op-node (concat 'times block1 '- block2 '-v *name-counter*))
          (create-op-node cyto-op-node (eval cyto-block-prod)
              cyto-block1 cyto-block2 50 (+ lev 1))
;Increments count.
          (setq *name-counter* (add1 *name-counter*))))))

;CREATE-CODERACK FUNCTION
;Initialize the coderack
(defun create-coderack ()
(cr-make-coderack 'my-coderack
     (list %upper-urgency% %first-urgency% %second-urgency%
           %third-urgency% %fourth-urgency% %fifth-urgency% 0))
(setq *coderack* 'my-coderack))
;CREATE-CYTO-NODE CODELET
(defun create-cyto-node (activation name value status
                         level type success)
;Codelet which creates a new node in the cytoplasm
;Loads a link-to-pnet codelet
;Name is the name of the variable which contains the address of the node
;Loads a compare-b-to-t codelet if the node type is "2b" or "4bl".
;Updates the context if the node type is "1t" or "3dt".
    (set name (make-instance 'cyto-node
                  :activation activation
                  :name name
                  :value value
                  :status status
                  :level level
                  :type type
                  :success success))
    (format t "Node ~a created" name) (terpri)
    (send *cytoplasm* :set-nodes (cons (eval name) (send *cytoplasm* :nodes)))
    (cr-hang *coderack* (list 'link-to-pnet name type value)
                        %upper-urgency%)
    (if (or (equal type "2b") (equal type "4bl")) then
      (cr-hang *coderack* (list 'compare-b-to-t name) %upper-urgency%))
    (if (or (equal type "1t") (equal type "3dt")) then
      (update-current-target (eval name) 100)))
```

```
;CREATE-OP-NODE CODELET
(defun create-op-node (name res op1 op2 activation level)
;Codelet which creates an operation node in the cytoplasm
;Creates links with the cyto-nodes res, op1, op2
;Updates the neighbors variable of res, op1, op2
;Updates the cytoplasm nodes variable.
     (setq n1 (list res 'result))
     (setq n2 (list op1 'operand))
     (setq n3 (list op2 'operand))
     (set name (make-instance 'cyto-node
                    :name name
                    :type "5g"
                    :level level
                    :neighbors (list n1 n2 n3)))
     (format t "Node ~a created" name) (terpri)
     (send *cytoplasm* :set-nodes (cons (eval name) (send *cytoplasm* :nodes)))
     (send res :update-neighbors (list (list (eval name) 'result)))
     (send op1 :update-neighbors (list (list (eval name) 'operand)))
     (send op2 :update-neighbors (list (list (eval name) 'operand))))

;DECOMP+ CODELET
(defun decomp+ (cyto-block cyto-current-target)
;Checks if the nodes are still free
;Creates a derived target in the cytoplasm
;Changes the status of the current-target to "linked"
;Changes the status of the block to "linked"
;Changes the current-target
;Create a node + in the cytoplasm. Increments count to distinguish
;possible different versions of operations and targets.
;Loads compare codelets for all the free blocks and bricks in the cyto.
(let (stblock sttarget lev block current-target
        res cyto-oper cyto-res cyto-derived-target diff cyto-op-node)
(setq stblock (send cyto-block :status))
(setq sttarget (send cyto-current-target :status))
(setq lev (send cyto-current-target :level))
(setq block (send cyto-block :value))
(setq current-target (send cyto-current-target :value))
(cond
     ((or (nequal "free" stblock) (nequal "free" sttarget)) nil)
     ((= block current-target)
          (cr-hang *coderack* (list 'replace-target cyto-block
                    cyto-current-target) %upper-urgency%))
     (t
     (cond
          ((> block current-target)
           (setq res block) (setq cyto-res cyto-block) (setq oper current-target)
           (setq cyto-oper cyto-current-target))
          ((<= block current-target)
           (setq oper block) (setq cyto-oper cyto-block) (setq res current-target)
           (setq cyto-res cyto-current-target)))
     (setq diff (- res oper))
     (setq cyto-derived-target (concat 'cyto-target- diff '-v *name-counter*))
;Creates a derived target in the cytoplasm
     (create-cyto-node 200 cyto-derived-target diff "free"
```

```
        (- lev 2) "3dt" 0)
;Changes the status of the current-target to "linked"
    (send cyto-current-target :set-status "linked")
;Changes the status of the block to "linked"
    (send cyto-block :set-status "linked")
;Changes the current-target
    (update-current-target (eval cyto-derived-target) 50)
;Creates a node + in the cytoplasm. Increments count to distinguish
;possible different versions of operations and targets.
    (setq cyto-op-node (concat 'plus oper '- diff '-v *name-counter*))
    (setq *name-counter* (+ 1  *name-counter*))
    (create-op-node   cyto-op-node cyto-res cyto-oper
        (eval cyto-derived-target) 50 (- lev 1))
;Loads compare codelets for all the free blocks and bricks in the cyto.
    (loop for nn in (send *cytoplasm* :cyto-brick-block-nodes) do
          (cond
               ((is-linked-to-target nn) nil)
               (t
    (cr-hang *coderack* (list 'compare-b-to-t nn) %first-urgency%))))))))

;DECOMPx CODELET
(defun decompx (cyto-block cyto-current-target)
;Checks if the nodes are still free
;Creates a derived target in the cytoplasm
;Changes the status of the current-target to "linked"
;Changes the status of the block to "linked"
;Changes the current-target
;Create a node + in the cytoplasm. Increments count to distinguish
;possible different versions of operations and targets.
;Loads compare codelets for all the free blocks and bricks in the cyto.
(let (stblock sttarget lev oper res cyto-oper
        cyto-res quot cyto-derived-target cyto-block cyto-op-node)
(setq stblock (send cyto-block :status))
(setq sttarget (send cyto-current-target :status))
(setq lev (send cyto-current-target :level))
(cond
    ( (or (nequal "free" stblock) (nequal "free" sttarget)) nil)
    (t
    (setq oper (send cyto-block :value))
    (setq res (send cyto-current-target :value))
    (setq cyto-oper cyto-block)
    (setq cyto-res cyto-current-target)
    (setq quot (/ res oper))
    (setq cyto-derived-target (concat 'cyto-target- quot '-v *name-counter*))
;Creates a derived target in the cytoplasm
    (create-cyto-node 200 cyto-derived-target quot "free"
        (- lev 2) "3dt" 0)
;Changes the status of the current-target to "linked"
    (send cyto-current-target :set-status "linked")
;Changes the status of the block to "linked"
    (send cyto-block :set-status "linked")
;Changes the current-target
    (update-current-target (eval cyto-derived-target) 50)
;Creates a node x in the cytoplasm. Increments count to distinguish
;possible different versions of operations and targets.
    (setq cyto-op-node (concat 'times oper '- quot '-v *name-counter*))
```

```
    (setq *name-counter* (add1 *name-counter*))
    (create-op-node   cyto-op-node cyto-res cyto-oper
        (eval cyto-derived-target) 50 (- lev 1))
;Loads compare codelets for all the free blocks and bricks in the cyto.
    (loop for nn in (send *cytoplasm* :cyto-brick-block-nodes) do
        (cond
            ((is-linked-to-target nn) nil)
            (t
    (cr-hang *coderack* (list 'compare-b-to-t nn) %first-urgency%)))))))

;DECOMPI CODELET
(defun decompi (cyto-block cyto-current-target rank-list)
;Loads a Look-for-blx codelet if the rank-list has a 1 in the first or
;second position, that is if the target is equal to 10 or 100 times
;the block plus something less than 10 or 100.
;The activation variable of the block is increased. This will make it
;more probably chosen when a new block will be built.
(let ((target (send cyto-current-target :value))
      (block (send cyto-block :value))
      (op2 nil))
    (cond
        ((= 1 block) nil)
        ((= 1 (cadr rank-list))
         (setq op2 10) (send cyto-block :set-activation 200))
        ((= 1 (car rank-list))
         (setq op2 100) (send cyto-block :set-activation 100)))
    (if op2 then
        (cr-hang *coderack* (list 'look-for-blx (times block op2)
                                  block op2) %second-urgency%))))


;DECOMPOSE FUNCTION
(defun decompose (node)
;This function expands a given node using its neighbors variable.
;Used in the decoding of the result.
;The operation node is first extracted and from it the relevant
;information is searched for and printed. The function is recursived.
;Once a operation node has been listed, its listed variable is put to t.
(let ((op) (opn) (close) (operands) (op1v) (op2v) (op1n) (op2n))
    (setq op (send (eval node) :neighbors))
;Considers all the neighbors (operation nodes) of the node.
(do ((x op (cdr x)))
    ((null x))
;Controls if the operation has not been already considered
(cond
    ((send (caar x) :listed) nil)
;If not yet considered it is decomposed
    (t
;The listed variable of the op node is put to t.
    (send (caar x) :set-listed t)
    (setq opn (send (caar x) :name))
    (setq close (send (caar x) :neighbors))
    (setq operands (sup close (eval node)))
    (setq op1v (send (car operands) :value))
    (setq op2v (send (cadr operands) :value))
    (setq op1n (send (car operands) :name))
    (setq op2n (send (cadr operands) :name))
```

```
    (format t "Operation ~a has been applied " opn)
    (terpri) (format t "to ~a ( ~a) and to ~a ( ~a)" opln oplv op2n op2v)
    (terpri) (format t "to get ~a" (send (eval node) :name)) (terpri)
    (if (nequal "2b" (send (eval opln) :type)) then (decompose opln))
    (if (nequal "2b" (send (eval op2n) :type)) then (decompose op2n)))))))

;DECREASE-INTEREST FUNCTION
(defun decrease-interest ()
;This function decreases (by 40%) the level of interest
;(activation variable) of the cyto-nodes which are free
;and secondary nodes.
(let ((list nil) (cyto-nodes nil))
    (setq cyto-nodes (send *cytoplasm* :free-secondary-cyto-nodes))
    (loop for node in cyto-nodes do
          (setq new (times (send node :activation) 0.6))
          (send node :set-activation new))))

;DIFF300 FUNCTION
(defun diff300 (val)
;used in check-temperature
(diff 300 val))

;DIGITS-IN-COMMON FUNCTION
;This function examines if two given values start with the same
;digit(s) or end with the same digit(s)
;Returns nil or a 3 dimensional vector.
;Examples: (digits-in-common 7 71) gives (0 1 0)
;Examples: (digits-in-common 7 607) gives (0 0 1)
;Examples: (digits-in-common 7 707) gives (1 0 1)
;Examples: (digits-in-common 22 222) gives (0 1 1)
(defun digits-in-common (val1 val2)
(let ((res '(0 0 0)))
  (cond
        ((> val1 val2) (setq res nil))
        (t
              (if (zerop (mod (- val2 val1) 10)) then (setq res '(0 0 1)))
              (if (equal val1 (*quo val2 10))
                          then (setq res (list 0 1 (caddr res))))
              (if (equal val1 (*quo val2 100))
                          then (setq res (cons 1 (cdr res)))
                          else res)))
  (cond
       ((equal res '(0 0 0)) (setq res nil))
       (t res))))

;DISCONNECT FUNCTION
(defun disconnect (cyto-node op)
;Suppress a given block/derived-target in the cytoplasm.
;Takes the neighbors of its op-neighbor and frees them.
;Updates all the neighbors variable.
;Updates the node variable of the cytoplasm.
;Suppresses the link in the pnet by loading an "free-from-pnet"
;codelet.
(let ((close) (operands) (opln) (op2n) (opn) (cyton))
;Frees the component blocks.
      (setq close (send op :neighbors))
```

```
          (setq operands (sup close cyto-node))
          (setq op1n (send (car operands) :name))
          (setq op2n (send (cadr operands) :name))
          (if (equal "linked" (send (eval op1n) :status))
          (send (eval op1n) :set-status "free"))
          (if (equal "linked" (send (eval op2n) :status))
          (send (eval op2n) :set-status "free"))
;Updates the neighbors variable.
          (send (eval op1n) :suppress-neighbors op)
          (send  (eval op2n) :suppress-neighbors op)
          (send cyto-node :suppress-neighbors op)
;Updates the node variable of the cytoplasm.
          (send *cytoplasm* :suppress-node op)
          (setq opn (send op :name))
          (format t "Node ~a killed" opn) (terpri)
          (send *cytoplasm* :suppress-node cyto-node)
          (send cyto-node :set-status "killed")
          (setq cyton (send cyto-node :name))
          (format t "Node ~a killed" cyton) (terpri)
;Suppresses the link in the pnet by loading an "free-from-pnet"
;codelet.
          (cr-hang *coderack* (list 'free-from-pnet cyto-node)
              %first-urgency%)))

;ELIMINATE FUNCTION
(defun eliminate (element list)
;Eliminates from a given list the member which is the closest from
;element. For instance (eliminate 8 '(4 6 7) ) should give (4 6).
;In case of ties, the first best is eliminated.
     (let ((diffmin 999))
     (do
          ((x list (cdr x)))
          ((null x))
          (setq diff (abs (- element (car x))))
          (cond
               ((< diff diffmin) (setq diffmin diff) (setq min (car x)))
               (t nil)))
     (remove-dd min list)))

;FIND-ACTIVATIONS FUNCTION
(defun find-activations (list)
;Finds the activations of a list of cyto-nodes.
(let ((res nil)(x nil))
    (do ((v list (cdr v)))
          ((null v) (reverse res))
          (setq x (send (car v) :activation))
          (setq res (cons x res)))))

;FIND-IN FUNCTION
(defun find-in (element list &optional (level 4))
;Finds in a list of nodes a node with a value similar to a given value.
;The level of similarity is optionally specified in level. Its meaning
;is relative (see sim function) , 1 being a perfect match.
;A node is randomly chosen (the weights are the activations of the nodes)
;in the list and its value is compared with
;element. If not too different the node is returned.
```

```
    (let((res1 list) (res2 (find-activations list)) (rep nil))
    (do
        ((x (randlist res2) (randlist res2)))
        ((equal nil x) rep)
        (setq rep (nth x res1))
        (cond
         ((> level (sim (send rep :value) element))
            (setq res2 nil))
         (t (setq res1 (remove-dd rep res1))
            (setq res2 (remove-dd (nth x res2) res2))
            (setq rep nil))))))
;FIND-INTEREST-IN-PNET FUNCTION
(defun find-interest-in-pnet (val)
;Finds the activation of the corresponding pnode in the pnet
;and returns an urgency. If the pnode is a target the urgency
;is maximal. If it is a brick the urgency is medium. In the other
;cases, the urgency is an increasing fonction of the activation.
(let (node type urgency priority)
 (cond
  ((< val 200)
   (setq node (find-node val))
   (setq priority (send node :activation))
   (setq type (caar (sortcar (send node :instances) nil)))
   (cond
        ((= val (send (send *current-target* :name) :value))
           (setq urgency %upper-urgency%))
        ((or (equal type "1t") (equal type "3dt"))
           (setq urgency %first-urgency%))
        ((and (nequal type "2b") (nequal type "4b1"))
           (cond
              ((< priority 10) (setq urgency %fifth-urgency%))
              ((< priority 14) (setq urgency %fourth-urgency%))
              ((< priority 50) (setq urgency %third-urgency%))
              (t (setq urgency %second-urgency%))))
        (t (setq urgency %third-urgency%))))
  ((< val 500) (setq urgency %fifth-urgency%))
  (t (setq urgency 0)))))

;FIND-NODE FUNCTION
(defun find-node (val)
;This function finds the pnode which should be associated to a given
;value (generally, the closest one)
;Similar to the code used in LINK-TO-PNET
(setq address (concat 'node- val))
(cond
    ((boundp address) t)
    ((= 0 val) (setq address 'node-1))
    ((< (round val) 200) (setq address (concat 'node- (round val))))
    (t (setq address 'node-150)))
(eval address))

;FREE-FROM-PNET CODELET
(defun free-from-pnet (name)
;Suppresses in the pnet the link to name.
;Decreases its activation.
```

```
(let ((pnode nil))
    (setq pnode (car (send name :plinks)))
    (cond
        (pnode (send pnode :suppress-instances name)
               (send pnode :add-activation %node-minus%)))))

;IS-LINKED-TO FUNCTION
(defun is-linked-to (cyto-block1 cyto-block2)
;Checks if two cyto-nodes are linked in the cytoplasm.
(let (lev1 lev2 st1 st2 node)
(setq lev1 (send cyto-block1 :level))
(setq lev2 (send cyto-block2 :level))
(setq st1 (send cyto-block1 :status))
(setq st2 (send cyto-block2 :status))
(cond
;If the levels are equal, they cannot be linked.
    ((= lev1 lev2)  nil)
;If they are both free, they cannot be free neither.
    ((and (equal "free" st1) (equal "free" st2)) nil)
;The level of cyto-block1 will be considered the lower.
;If it is not the case they are interchanged.
    (t
      (when (> lev1 lev2)
        (setq node cyto-block2)
        (setq cyto-block2 cyto-block1)
        (setq cyto-block1 node))
;If the lower cyto-block (that is cyto-block1) is free,
;once again, they cannot be linked.
      (cond
          ((nequal "linked" (send cyto-block1 :status)) nil)
          (t
;The upper neighbors of cyto-block1 are examined. If one
;of them is equal to cyto-block2, then the function will
;return "true", if not, it will return "nil".
           (do
              ((node (send (send cyto-block1 :upper-neighbor)
                           :upper-neighbor)
                     (send (send node :upper-neighbor)
                           :upper-neighbor)))
              ((null node) nil)
              (cond
                   ((eq node cyto-block2)
                    (setq node nil) (return t))
                   ((equal "free" (send node :status))
                    (return nil))
                   ((equal "1t" (send node :type))
                    (return nil))
                   (t nil)))))))))

;IS-LINKED-TO-TARGET FUNCTION
(defun is-linked-to-target (cyto-block)
;This function checks if a given block/brick is linked in some way
;to the target.
(cond
    ((equal "linked" (send cyto-block :status))
 (do
```

```
  ((node cyto-block
        (send (send node :upper-neighbor) :upper-neighbor)))
   ((null node) nil)
   (cond
        ((equal "free" (send node :status)) (return nil))
        ((equal "1t" (send node :type)) (return t))
        (t nil))))))

;KILL-BLOCK FUNCTION
(defun kill-block (cyto-block)
;This function is able to suppress a cyto-block in the tree structure of
;the cytoplasm. Two different cases have to be considered. First the node
;can be a free block. Then, we just have to disconnect the two blocks or
;bricks which are under it in the tree (they are linked through an
;operation node of course). If the node is not free, two different
;manipulations are needed. First, as in the first case, disconnection of
;the children but also determination of the parent and suppression of
;that parent by recursion. If the parent happens to be the target, a call
;to kill-dtarget is needed. It will proceed by going down in the tree,
;from parent to children (one of whose is always a derived-target).
;
;Disconnects the cyto-block from the nodes which are under it.
(let (op cyto-node)
(disconnect cyto-block (send cyto-block :lower-neighbor))
(cond
;If the block was free (thus no more neighbors), it is finished.
    ((null (send cyto-block :neighbors)) nil)
;If the block was not free, it calls op the upper operation-node.
    (t
        (setq op (caar (send cyto-block :neighbors)))
;It determines the upper block and if it is not the target, applies
;recursively kill-block. If it is the target, kill-dtarget is applied
;to the first derived-target (child of the target).
        (setq cyto-node (send op :upper-neighbor))
        (cond
            ((equal "4bl" (send cyto-node :type)) (kill-block cyto-node))
            ((equal "1t" (send cyto-node :type))
             (setq cyto-node (send op :lower-dtarget-neighbor))
             (kill-dtarget cyto-node)))))))

;KILL-DTARGET FUNCTION
(defun kill-dtarget (cyto-dtarget)
;This function is able to suppress a cyto-derived-target in the tree structure of
;the cytoplasm. Two different cases have to be considered. First the node
;can be a free derived-target. Then, we just have to disconnect the two nodes
;(block/brick and dtarget/target) which are above and next to it in the tree
;(they are linked through an operation node of course).
;If the node is not free, two different manipulations are needed.
;First, as in the first case, disconnection of the parent and the brother
;but also determination of the dtarget-child and suppression of it by
;recursion
;
;Disconnects the cyto-dtarget
(disconnect cyto-dtarget (send cyto-dtarget :upper-neighbor))
(let (op cyto-node)
(cond
```

```
;If the derived-target was free (no more neighbors), it is finished.
    ((null (send cyto-dtarget :neighbors)) nil)
;If the derived-target was not free, it calls op the lower operation-node.
    (t
        (setq op (caar (send cyto-dtarget :neighbors)))
;It determines the dtarget-child and applies recursively kill-dtarget.
        (setq cyto-node (send op :lower-dtarget-neighbor))
        (kill-dtarget cyto-node)))))

;KILL-NODE CODELET
(defun kill-node (cyto-node)
;This codelet suppresses a block or a derived-target in the cytoplasm
;and all the dependent nodes.
;Depending on the type of the node, it uses kill-block or kill-dtarget.
;It updates the current-target and reactivates the corresponding
;pnode in the pnet (and the result pnode).
(let ((type) (pnode nil) (new-target nil) old-block)
  (cond
   ((memq cyto-node (send *cytoplasm* :nodes))
    (setq type (send cyto-node :type))
    (cond
        ((equal type "4bl") (kill-block cyto-node))
        ((equal type "3dt")
;If the node to be killed is a target, its old block neighbor is
;saved in order to avoid to recreate the same association directly
;after.
         (setq old-block (send cyto-node :block-neighbor))
         (kill-dtarget cyto-node))
        (t nil))
    (setq new-target (send *cytoplasm* :find-new-target))
    (update-current-target new-target 100)
    (if new-target
    (setq pnode (car (send (send *current-target* :name) :plinks))))
    (if pnode
      (cr-hang *coderack* (list 'activate %dtarget-plus% pnode t)
        %upper-urgency%))
;Loads compare codelets for all the free blocks and bricks in the cyto.
    (loop for nn in (send *cytoplasm* :cyto-brick-block-nodes) do
          (cond
               ((is-linked-to-target nn) nil)
               ((eq old-block nn) nil)
               (t
   (cr-hang *coderack* (list 'compare-b-to-t nn) %second-urgency%))))))))

;LINK-TO-PNET CODELET
(defun link-to-pnet ( name type value)
;Determines the level of activation depending on the type of node.
;If the node is a derived-target, the result node in the pnet is
;reactivated via the loading of an activate codelet.
;Updates instances in the closest node of the pnet
;Loads an activate codelet
    (let ((repump-yes-or-no nil) (address nil) (transact nil))
(cond ((equal "killed" (send (eval name) :status)) nil)
   (t
    (cond
        ((equal type "1t") (setq activation %target-activation%)
```

```
          (setq repump-yes-or-no t))
         ((equal type "2b") (setq activation %brick-activation%))
         ((equal type "3dt") (setq activation %dtarget-activation%)
          (setq repump-yes-or-no t))
         ((equal type "4bl") (setq activation %block-activation%)))
    (setq address (concat 'node- value))
    (cond
        ((boundp address) (setq transact activation))
        ((= 0 value) (setq address 'node-1)
                     (setq transact 0))
        ((< (round value) 200)
           (setq address (concat 'node- (round value)))
           (setq transact (times activation (ratio value))))
        (t (setq address 'node-150)
           (setq transact (times activation 0.50))))
    (send (eval address) :update-instances (list type name))
    (send  name :update-plinks (eval address))
    (cr-hang *coderack* (list 'activate transact
                          address repump-yes-or-no) %upper-urgency%)))))

;LOOK-FOR-APPROX-BL+ CODELET
(defun look-for-approx-bl+ (res op1 op2)
;Given 3 numbers which represent an operation (res = op1 + op2)
;checks if this type of operation could be instantiated in the
;cytoplasm.
;Checks first if one of the 3 arguments is similar to ther
;current-target.
;Then eliminates in (res op1 op2) the value closest to the current-
;target.
(let (current-target values-to-find blocks rn first-value
         cyto-block1 cyto-block2 n-blocks second-value v1 v2
         sum)
   (setq current-target (send (eval (send *current-target* :name)) :value))
(cond
    ((compare current-target (list res op1 op2))
   (setq values-to-find (eliminate current-target (list res op1 op2)))
;Tries to find randomly a block not too far from one of the given
;value. In order to do that, chooses randomly a value and a block and
;compares them.
   (setq blocks (send *cytoplasm* :cyto-brick-block-nodes))
   (setq rn (random 2))
   (setq first-value (nth rn values-to-find))
   (setq cyto-block1 (find-in first-value blocks))
;Tries to find another block not too far from the second value.
   (setq n-blocks (remove-dd cyto-block1 blocks))
   (setq second-value (nth (- 1 rn) values-to-find))
   (setq cyto-block2 (find-in second-value n-blocks))
;Checks if the operation should be implemented in the cytoplasm or if
;a derived target has to be defined.
;If one of the cyto-block is absent, a decomp+ codelet is loaded.
   (cond
        ((and (null cyto-block1) (null cyto-block2)) nil)
        ((null cyto-block1)
         (cr-hang *coderack* (list 'decomp+ cyto-block2
            (send *current-target* :name)) %second-urgency%))
        ((null cyto-block2)
```

```
        (cr-hang *coderack* (list 'decomp+ cyto-block1
          (send *current-target* :name)) %second-urgency%))
;If none is absent, a const-bl+ codelet is loaded.
       (t
        (setq v1 (send cyto-block1 :value))
        (setq v2 (send cyto-block2 :value))
;Determines the exact operation to implement.
        (cond
          ((member res values-to-find) (setq sum (abs (- v1 v2))))
          (t (setq sum (+ v1 v2))))
;Determines the interest and the feasability of creating the new block
;by loading a "test-if-possible-and-desirable codelet".
          (cr-hang *coderack* (list 'test-if-possible-and-desirable
           cyto-block1 cyto-block2 "const-bl+" sum) %second-urgency%)))))))
;old method to load the codelet
;        (cond
;              ((< (sim sum current-target) 4)
;                    (cr-hang *coderack* (list 'const-bl+ cyto-block1 cyto-blo
;                      sum) %second-urgency%))
;              (t nil)))

;LOOK-FOR-APPROX-BLX CODELET
(defun look-for-approx-blx (res op1 op2)
;Given 3 numbers which represent an operation (res = op1 x op2)
;checks if this type of operation could be instantiated in the
;cytoplasm.
;First eliminates in (res op1 op2) the value closest to the current-
;target.
(let (current-target values-to-find blocks rn first-value cyto-block1
      cyto-block2 n-blocks second-value v1 v2 prod)
(setq current-target (send
      (eval (send *current-target* :name)) :value))
(cond
   ((compare current-target (list res op1 op2))
   (setq values-to-find (eliminate current-target (list res op1 op2)))
;Tries to find randomly a block not too far from one of the given
;value. In order to do that, chooses randomly a value and a block and
;compares them.
   (setq blocks (send *cytoplasm* :cyto-brick-block-nodes))
   (setq rn (random 2))
   (setq first-value (nth rn values-to-find))
   (setq cyto-block1 (find-in first-value blocks))
;Tries to find another  block not too far from the second value.
   (setq n-blocks (remove-dd cyto-block1 blocks))
   (setq second-value (nth (- 1 rn) values-to-find))
   (setq cyto-block2 (find-in second-value n-blocks))
;Checks if the operation should be implemented in the cytoplasm.
;If it is the case loads a const-blx codelet.
   (cond
        ((or (null cyto-block1) (null cyto-block2)) nil)
        (t
         (setq v1 (send cyto-block1 :value))
         (setq v2 (send cyto-block2 :value))
         (setq prod (times v1 v2))
;Determines the interest and the feasability of creating the new block
```

```
;by loading a "test-if-possible-and-desirable codelet".
          (cr-hang *coderack* (list 'test-if-possible-and-desirable
          cyto-block1 cyto-block2 "const-blx" prod) %second-urgency%))))))
;Old method
;          (cond
;
;                ((< (sim prod current-target) 4)
;                      (cr-hang *coderack* (list 'const-blx cyto-block1 cyto-blo
;                         prod) %second-urgency%))
;                (t nil)))))

;LOOK-FOR-BL+ CODELET
(defun look-for-bl+ (res op1 op2)
;Given 3 numbers which represent an operation (res = op1 + op2)
;checks if this operation could be exactly instantiated in the
;cytoplasm.
;First eliminates in (res op1 op2) the value closest to the current-
;target.
(let (current-target values-to-find node1 node2 cyto-block1
        cyto-block2 v1 v2 sum st1 st2)
   (setq current-target (send (eval (send *current-target* :name)) :value))
   (setq values-to-find (eliminate current-target (list res op1 op2)))
;Tries to reach the target with the cyto-nodes linked to the
;pnodes corresponding to values-to-find (here op1 and op2 but could
;be more general)
   (setq node1 (find-node (car values-to-find)))
   (setq node2 (find-node (cadr values-to-find)))
   (setq cyto-block1 (cadar (send (eval node1) :instances)))
;Here I use a kludge to take into account the fact that we can have
;op1 = op2
   (setq cyto-block2 (cadar (reverse (send (eval node2) :instances))))
;Checks if cyto-block1 and cyto-block2 exist
(cond
    ((or (null cyto-block1) (null cyto-block2) (eq cyto-block1 cyto-block2))
     (cr-hang *coderack* (list 'look-for-approx-bl+ res op1 op2)
         %first-urgency%))
    ((or (equal "1t" (send cyto-block1 :type))
         (equal "3dt" (send cyto-block1 :type)))
     (cr-hang *coderack* (list 'look-for-approx-bl+ res op1 op2)
         %first-urgency%))
    ((or (equal "1t" (send cyto-block2 :type))
         (equal "3dt" (send cyto-block2 :type)))
     (cr-hang *coderack* (list 'look-for-approx-bl+ res op1 op2)
         %first-urgency%))
    (t
      (setq v1 (send cyto-block1 :value))
      (setq v2 (send cyto-block2 :value))
;Determines the exact operation to implement
      (cond
          ((= res (car values-to-find)) (setq sum (- v1 v2)))
          ((= res (cadr values-to-find)) (setq sum (- v2 v1)))
          (t (setq sum (+ v1 v2))))
      (cond
          ((nequal current-target sum)
     (cr-hang *coderack* (list 'look-for-approx-bl+ res op1 op2)
         %first-urgency%))
          (t
```

```
;Loads a "test-if-possible-and-desirable codelet" with a very high
;urgency, since the block is equal to the target.
            (cr-hang *coderack*
                (list 'test-if-possible-and-desirable
                      cyto-block1 cyto-block2 "const-bl+" sum)
          %upper-urgency%)))))))

;LOOK-FOR-BLX CODELET
(defun look-for-blx (res op1 op2)
;Given 3 numbers which represent an operation (res = op1 + op2)
;checks if this operation could be exactly instantiated in the
;cytoplasm.
;First eliminates in (res op1 op2) the value closest to the current-
;target.
   (setq current-target (send (eval (send *current-target* :name)) :value))
   (setq values-to-find (eliminate current-target (list res op1 op2)))
;Tries to reach the target with the cyto-nodes linked to the
;pnodes corresponding to values-to-find (here op1 and op2 but could
;be more general)
   (setq node1 (find-node (car values-to-find)))
   (setq node2 (find-node (cadr values-to-find)))
   (setq cyto-block1 (cadar (send (eval node1) :instances)))
;Here I use a kludge to take into account the fact that we can have
;op1 = op2
   (setq cyto-block2 (cadar (reverse (send (eval node2) :instances))))
;Checks if cyto-block1 and cyto-block2 exist
(cond
    ((or (null cyto-block1) (null cyto-block2) (eq cyto-block1 cyto-block2))
     (cr-hang *coderack* (list 'look-for-approx-blx res op1 op2)
         %first-urgency%))
    ((or (equal "1t" (send cyto-block1 :type))
         (equal "3dt" (send cyto-block1 :type)))
     (cr-hang *coderack* (list 'look-for-approx-blx res op1 op2)
         %first-urgency%))
    ((or (equal "1t" (send cyto-block2 :type))
         (equal "3dt" (send cyto-block2 :type)))
     (cr-hang *coderack* (list 'look-for-approx-blx res op1 op2)
         %first-urgency%))
    ((nequal current-target
         (times (send cyto-block1 :value) (send cyto-block2 :value)))
     (cr-hang *coderack* (list 'look-for-approx-blx res op1 op2)
         %first-urgency%))
    (t
;Loads a "test-if-possible-and-desirable codelet" with a very high
;urgency, since the block is equal to the target.
            (cr-hang *coderack*
                (list 'test-if-possible-and-desirable
                      cyto-block1 cyto-block2 "const-blx" current-target)
          %upper-urgency%))))


;LOOK-FOR-DIFF CODELET
(defun look-for-diff (trial)
;Tries to combine similar bricks to get small numbers. Trial is
;the number of trials already made.
(let (current-target blocks rn cyto-block1 value1 n-blocks
```

```
        (cyto-block2 nil) node urgency value2)
    (setq current-target (send (eval (send *current-target* :name))
                               :value))
;Chooses randomly a  block which will be called cyto-block1
;If trial = 0, just the free blocks are considered. If trial > 0
;all the blocks are taken into account
    (if (= 0 trial)
    (setq blocks (send *cytoplasm* :free-blocks))
    (setq blocks (send *cytoplasm* :cyto-brick-block-nodes)))
(cond ((null blocks) nil)
   (t
    (setq rn (random (length blocks)))
    (setq cyto-block1 (nth rn blocks))
    (setq value1 (send cyto-block1 :value))
;Updates blocks by removing cyto-block1
    (setq n-blocks (remove-dd cyto-block1 blocks))
;Tries to find another free block similar to cyto-block1 (cyto-block2)
    (do
       ((x n-blocks n-blocks))
       ((null x) cyto-block2)
       (setq node (nth (random (length n-blocks)) n-blocks))
       (setq similarity (sim value1 (send node :value)))
       (cond
             ((< 3 similarity) (setq n-blocks
                                     (remove-dd node n-blocks)))
             (t (setq n-blocks nil) (setq cyto-block2 node))))
;Checks if cyto-block2 exists. If it is not the case, a new
;look-for-diff codelets is loaded, if trial is less than 5.
;The urgency of that codelet is function of trial.
    (cond
         ((and (null cyto-block2) (< trial 4))
          (cond
               ((> 2 trial) (setq urgency %first-urgency%))
               ((> 5 trial) (setq urgency %second-urgency%)))
          (setq trial (add1 trial))
          (cr-hang *coderack* (list 'look-for-diff trial) urgency))
;If cyto-block2 exists, a look-for-bl+ codelets is loaded
         (cyto-block2
             (setq value2 (send cyto-block2 :value))
             (cr-hang *coderack* (list 'test-if-possible-and-desirable
             cyto-block1 cyto-block2 "const-bl+" (abs (- value1 value2)))
                    %first-urgency%))))))

;LOOK-FOR-NEW-BLOCK CODELET
(defun look-for-new-block ()
;This codelet tries to build a new block by randomly choosing two
;bricks or blocks in the cytoplasm and combining them. The operation
;used to combine them depends on the activitions of the nodes "add",
;"subtract" and "multiply" in the Pnet. A random choice is performed
;with probabilities proportional to those activations.
;Once a new block has been formed, it is tested by
;"test-if-possible-and-desirable".
(let (blocks list rank cyto-block1 cyto-block2 res  node activation
              type  w+ w- wx fun)
;Random choice of two blocks according to their level of interest
;measured by their activation stored in liste.
```

```
    (setq blocks (send *cytoplasm* :cyto-brick-block-nodes))
    (setq liste (find-activations blocks))
    (setq rank (randlist liste))
    (setq cyto-block1 (nth rank blocks))
    (setq blocks (remove-dd cyto-block1 blocks))
    (setq liste (remove-dd (nth rank liste) liste))
    (setq rank (randlist liste))
    (setq cyto-block2 (nth rank blocks))
;Random choice of an operation according to the level of activation
;of the corresponding pnodes.
    (setq w+ (send node-add :activation))
    (setq w- (send node-subtract :activation))
    (setq wx (send node-multiply :activation))
    (setq rank (randlist (list w+ w- wx)))
    (cond
      ((= 0 rank)
       (setq res (+ (send cyto-block1 :value) (send cyto-block2 :value)))
       (setq fun  "const-bl+"))
      ((= 1 rank)
       (setq res (abs (- (send cyto-block1 :value)
                         (send cyto-block2 :value))))
       (setq fun "const-bl+"))
      (t (setq res (* (send cyto-block1 :value)
                      (send cyto-block2 :value)))
         (setq fun "const-blx")
;If one of the blocks is equal to 1 the operation must be
;abandoned. This is achieved by setting res to 0.
         (if (or (= 1 (send cyto-block1 :value))
                 (= 1 (send cyto-block2 :value))) (setq res 0))))
;Loading of the "test-if-possible-and-desirable codelet"
         (cr-hang *coderack*
          (list 'test-if-possible-and-desirable cyto-block1 cyto-block2
            fun res) %first-urgency%)))

;MEAN FUNCTION
(defun mean (list)
;this function computes the mean of a list of numbers.
(let ((sum 0))
  (cond
    ((null list) 0)
    (t
       (setq sum (apply 'add list))
       (quotient sum (length list))))))


;MISFORTUNE FUNCTION
(defun misfortune (interest level status)
;This function computes the misfortune (of a cyto-node) given
;the level of interest (maximum = 300)
;the level in the tree structure (for blocks the level is 3,5 or more
;and for a dtarget the level is 97,95 or less)
;the status (free or linked).
;The function used is (/ 20 interest)
                      + 10 (if status is "free")
;
(let ((st 0))
  (if (equal "free" status) then (setq st 10))
```

```
    (plus (quotient 20 interest) st)))

;MULTIPLE FUNCTION
;This function tests if val2 is a multiple of val1.
;Val1 must be different from 1.
(defun multiple (val1 val2)
    (cond
        ((>= val1 val2) nil)
        ((equal 1 val1) nil)
        ((zerop (mod val2 val1)) t)
        (t nil)))

;PROPAGATE-SUCCESS CODELET
(defun propagate-success ()
;This codelet seraches in the cytoplasm for operation nodes.
;If an operation node has two neighbors with success = 1, the success
;variable of the third one is put to 1. It loops until changes are not
;possible any more.
     (setq nn (send *cytoplasm* :nodes))
     (setq cont 1)
     (do ((x 1 (add1 x)))
         ((equal 0 cont))
         (setq cont 0)
         (loop for node in nn do
               (cond
                    ((equal "5g" (send node :type))
                     (setq n (update-success (send node :neighbors))))
                    (t (setq n nil)))
               (cond
                    ((null n) nil)
                    ((equal "1t" (send n :type))
                     (setq *problem-solved* 1) (setq cont 0))
                    (t (send n :set-success 1) (setq cont 1))))))

;RANDLIST FUNCTION
(defun randlist (list)
;Chooses randomly a rank between 0 and length list - 1 according to weights
;given in list. For instance (randlist '(100 50)) should give 0 with a
;probability 2/3 and 1 with a probability 1/3.
      (cond
       ((null list) nil)
       ((< 0 (fix (apply 'add list)))
        (let ((r (random (fix (apply 'add list)))) (sum 0))
           (do ((v list (cdr v))
                (i -1 (add1 i)))
               ((null v) i)
               (setq sum (add sum (car v)))
               (cond
                    ((< r sum) (setq v nil))
                    (t nil)))))
       (t nil)))

;RATIO FUNCTION
(defun ratio (num)
;Gives the level to which the activation has to be decreased
;   if no node exists in the pnet
```

```
;    (cond
;       ((< (abs (- num (round num))) 5) 0.90)
;       ((< (abs (- num (round num))) 10) 0.80)
;       (t 0.70)))
    (quotient (float (min num (round num))) (float (max num (round num)))))

;READ-BRICK CODELET
(defun read-brick (i)
;Reads the brick i in the cytoplasm
;Loads a create-cyto-node codelet
;Loads a compare-b-to-t codelet
    (setq bricki  (concat 'brick i))
    (setq a  (intern (uconcat "brick" i) (find-package "keyword")))
    (setq brick (send *cytoplasm* a))
    (setq activation 50)
    (if (= 0 (mod brick 10)) (setq activation 300))
    (update-context "2b" (concat 'cyto-brick i) brick 50 "cyto")
    (setq cyto-bricki (concat 'cyto-brick i))
    (cr-hang *coderack* (list 'create-cyto-node activation
     (list 'quote cyto-bricki) brick "free" 1 "2b" 1) %first-urgency%))

;READ-TARGET CODELET
(defun read-target ()
;Read the target in the cytoplasm
;Update the context
;Load a create-cyto-node codelet
     (setq target (send *cytoplasm* :target))
     (update-current-target 'cyto-target 100)
;     (update-context "1t" 'cyto-target target 100 "cyto")
     (cr-hang *coderack* (list 'create-cyto-node 150 ''cyto-target target "free" 9
                               "1t" 0)
                               %first-urgency%))


;REMOVE-DD FUNCTION
(defun remove-dd (l1 l)
;Variant of the remove function which, in case of ex-aequo, remove just
;one element. (remove-dd 3 '(2 3 4 3)) should give (2 4 3). If nothing is
;found in the list the list is reversed.
    (let ((res nil))
    (do ((x l (cdr x)))
        ((null x) res)
        (cond
             ((eq (car x) l1)
              (setq res (append (reverse res) (cdr x))) (setq x nil))
             (t (setq res (cons (car x) res)))))))

;REPLACE-TARGET CODELET
(defun replace-target (cyto-block cyto-current-target)
;This codelet checks if the current-target is the target. If it is the
;case the problem is solved and the variable *problem-solved* is given the
;value 1.
;The cyto-block status is put to "linked".
;If it is not the case, the cyto-node corresponding to the current-target
;is replaced everywhere by th cyto-node corresponding to the given block.
;In order to do that, the neighbors of cyto-block must be updated by
```

```
;the neighbors of the current-target to suppress . The neighbors of
;the neighbors of the current-target must be updated as well.Idem for the
;pnodes which were linked to the current-target.
;The current-target is changed to the final target in the current-target flav.
;The current-target must be suppressed from the list of nodes in the
;*cytoplasm* flavour.
;Propagate the success.
   (cond
        ((equal (send cyto-target :value)
                (send cyto-block :value)) (setq *problem-solved* 1)
                                          (setq cyto-target cyto-block)
                                          (format t "Obvious. "))
        ((eq cyto-current-target (send *current-target* :name))
;The cyto-block status is put to "linked".
         (send cyto-block :set-status "linked")
;Update of the neighbors of the block
         (setq nn (send cyto-current-target :neighbors))
         (send  cyto-block :update-neighbors nn)
;Update of the neighbors of the neighbors
         (loop for pair in nn do
               (send (car pair) :replace-neighbors
                     cyto-current-target cyto-block))
;Update of the instances of the pnodes which are in the plinks of the cur-targ.
         (setq pp (send cyto-current-target :plinks))
         (loop for pnode in pp do
               (send pnode :suppress-instances cyto-current-target))
;Update of the cyto-current-target
         (update-current-target 'cyto-target 100)
;Update of the nodes variable in the *cytoplasm*
         (send *cytoplasm* :suppress-node cyto-current-target)
;Loading of the propagate codelet
         (propagate-success))))

;REPUMP FUNCTION
(defun repump ()
;Reactivates the operand, similar and result pnodes.
;Computes the values of %result+%, %resultx% and %operand% as
;a function of the activations of node-multiply and node-add +
;node-subtract.
  (let (resx res+)
       (setq resx (send node-multiply :activation))
       (setq res+ (diff 60 resx))
       (setq %resultx% (times 300 (quotient resx 60.0)))
       (setq %result+% (times 100 (quotient res+ 60.0)))
       (setq %operand% (quotient %resultx% 1.5))
       (set-up-activations (list 'result+ %result+%
                                 'resultx %resultx% 'operand %operand%
 'similar %similar% 'operation %operation% 'instance %instance%))))

;ROUND FUNCTION
(defun round (num)
;Gives the multiple of 5 which is closest to num if num < 20
;Gives the multiple of 10 which is closest to num if num < 100
;Otherwise gives the closest multiple of 50
       (cond
            ((< num 20)  (setq div 5))
```

```
          ((< num 100) (setq div 10))
          (t (setq div 50)))
       (cond
          ((< (*mod num div) 0) (times div (+ 1 (/ num div))))
          (t (times div (/ num div)))))

;SIM FUNCTION
;This functions computes the similarity between two values
;It will have to be replaced by something more general based
;on common descriptive characteristics.
(defun sim (val1 val2)
    (setq diff (abs (- val1 val2)))
    (setq diffrel (quotient (float diff) (float val2)))
    (cond
         ((= 0 diff) 0)
         ((<= diffrel 0.1) 1)
         ((<= diffrel 0.2) 2)
         ((<= diffrel 0.3) 3)
         (t 4)))

;SUP FUNCTION
(defun sup (l l1)
;l is a list of pairs. The pairs which start with l1 are suppressed.
;The result is the list of the first elements of the pairs not suppressed.
;Used in the decoding of the result.
    (let ((res nil))
    (do ((x l (cdr x)))
        ((null x) res)
        (cond
             ((eq (caar x) l1) nil)
             (t (setq res (cons (caar x) res)))))))

;TEMPERATURE FUNCTION
(defun temperature ()
;This functions computes the temperature of the cytoplasm.
;The temperature is based on the number of available free
;nodes (nfree), on the degree of elaboration of the current-target,
;on the average degree of misfortune of the nodes
;and on the maximum misfortune.
;The function used is
; (max misfortune) + 2 * (mean misfortune) if the number of free nodes
;plus the level of the current target ((99 - level)/2, in fact) is > 4
;if not, the function is
; (6 - nfree - lev) * (max misfortune) + 2 * (mean misfortune).
(let ((misfort nil) (nfree 0) (lev 0))
    (setq misfort (collect-misfortune))
    (setq nfree (length (send *cytoplasm* :free-cyto-nodes)))
    (setq lev (/ (- 99 (send (send *current-target* :name) :level)) 2))
    (cond
         ((> (add nfree lev) 4)
          (add (apply 'max misfort) (times 2 (mean misfort))))
         (t
          (add (times (- 6 nfree lev) (apply 'max misfort))
               (times 2 (mean misfort)))))))

;TEST-IF-POSSIBLE-AND-DESIRABLE CODELET
```

```
(defun test-if-possible-and-desirable (cyto-block1 cyto-block2 fun res)
;Checks if it is possible to agregate cyto-block1 and cyto-block2
;Aren't they linked to the target or to each other?
;If not, the interest of creating a new block (with the value res) is
;examined with the function "find-interest-in-pnet". If the interest
;is very high, the two blocks are freed and the fun codelet is loaded.
;If the interest is not very high, the fun codelet is tried.
;The value of fun is 'const+' or 'constx'.
(let (urgency (const 'const-blx) node)
;Checks if the blocks are not linked (one to the other)
;and not linked to the target.
 (cond
      ((is-linked-to cyto-block1 cyto-block2) (setq urgency nil))
      ((= res (send cyto-target :value)) (setq urgency %upper-urgency%))
      ((is-linked-to-target cyto-block1) (setq urgency nil))
      ((is-linked-to-target cyto-block2) (setq urgency nil))
;If res = 0 the operation is without any interest.
      ((= 0 res) (setq urgency nil))
;Computes the interest of the new possible block
      (t (setq urgency (find-interest-in-pnet res))))
;If the interest is high, cleans what must be cleaned.
;If a block is not free,
;this block is freed by killing its upper-neighbor, if its
;value is not equal to res (to avoid to kill and then recreate
;the same block).
  (cond
;If the urgency is nil, nothing must be done.
      ((null urgency) nil)
      (t
          (when
              (>= urgency %first-urgency%)
            (cond ((equal "linked" (send cyto-block1 :status))
                   (setq node
                      (send (send cyto-block1 :upper-neighbor)
                                 :upper-neighbor))
                   (if (nequal res (send node :value))
                      (kill-node node))))
            (cond ((equal "linked" (send cyto-block2 :status))
                   (setq node
                      (send (send cyto-block2 :upper-neighbor)
                                 :upper-neighbor))
                   (if (nequal res (send node :value))
                      (kill-node node)))))
;A fun codelet is loaded to construct the new block
          (if (equal fun "const-bl+") (setq const 'const-bl+))
          (cr-hang *coderack*
              (list const cyto-block1 cyto-block2 res)
          urgency)))))


;UPDATE-SUCCESS FUNCTION
(defun update-success (l)
;returns in a list of the three neighbors of an operation node (type "5g")
;the node with success = 0 if for the two other ones success = 1
;Used in the propagate-success codelet.
(let (n1 n2 n3 s1 s2 s3)
```

```
    (setq n1 (caar l))
    (setq n2 (caadr l))
    (setq n3 (caaddr l))
    (setq s1 (send n1 :success))
    (setq s2 (send n2 :success))
    (setq s3 (send n3 :success))
    (cond
        ((nequal 2 (plus s1 s2 s3)) nil)
        ((equal 0 s1) n1)
        ((equal 0 s2) n2)
        ((equal 0 s3) n3))))
```

```
(defvar *print-array*)
(defvar *coderack*)
(defvar *current-target*)
(defvar %initial-activation% 0.0)
(defvar %min-activation-to-be-added% 24.0)
(defvar %max-activation-to-be-transmitted% 35.0)
(defvar %k% 0.001)
(defvar %length% 1000)
(defvar %target-activation% 180)
(defvar %brick-activation% 60)
(defvar %dtarget-activation% 100)
(defvar %block-activation% 30)
(defvar %target-plus% 75)
(defvar %brick-plus% 25)
(defvar %dtarget-plus% 50)
(defvar %node-minus% -25)
(defvar %similar% 50)
(defvar %operation% 200)
(defvar %instance% 200)
(defvar %verbose% nil)
(defvar %graphics% nil)
(defvar %first-decay-rate% 0.50)
(defvar %second-decay-rate% 0.50)
(defvar %third-decay-rate% 0.70)
(defvar %fourth-decay-rate% 0.90)
(defvar %fifth-decay-rate% 0.90)
(defvar %sixth-decay-rate% 0.0)
(defvar %upper-threshold% 90)
(defvar %first-threshold% 30)
(defvar %upper-urgency% 600)
(defvar %first-urgency% 300)
(defvar %second-urgency% 70)
(defvar %third-urgency% 7)
(defvar %fourth-urgency% 4)
(defvar %fifth-urgency% 1)
(defvar %temperature-threshold% 80)
(defvar *name-counter* 1)

;Initialization of the pnet
(defun init-chiffre ()
    (setq *print-array* nil) ; don't print circular vectors
    (setq %initial-activation% 0.0)
    (setq %min-activation-to-be-added% 24.0)
    (setq %max-activation-to-be-transmitted% 35.0)
    (setq %k% 0.001)
    (setq %length% 1000)
    (setq %target-activation% 170)
    (setq %brick-activation% 60)
    (setq %dtarget-activation% 130)
    (setq %block-activation% 30)
    (setq %target-plus% 75)
    (setq %brick-plus% 40)
    (setq %dtarget-plus% 30)
    (setq %node-minus% -25)
    (setq %similar% 50)
    (setq %operation% 200)
```

```
    (setq %instance% 200)
    (setq %verbose% nil)
    (if (> (string-length (getenv "WINDOW_GFX")) 0)
        (setq %graphics% t)
        (setq %graphics% nil))
    (if %graphics%
        (format t "~&Graphics is ON.~%")
        (format t "~&Graphics is OFF.~%"))
    (setq %first-decay-rate% 0.50)
    (setq %second-decay-rate% 0.50)
    (setq %third-decay-rate% 0.70)
    (setq %fourth-decay-rate% 0.90)
    (setq %fifth-decay-rate% 0.90)
    (setq %sixth-decay-rate% 0.0)
    (setq %upper-threshold% 60)
    (setq %first-threshold% 30)
    (setq %temperature-threshold% 200)
    (initialize-pnet-2)
    (compile-flavor-methods pnode)
;Here the nodes which represent the link-types have received a pseudo
;instantiation in the cyto. Those nodes are not going to spread their
;activity. But they are not going to lose their level of activity as
;well because their decay-rate will be 0.0. This means that they will
;keep at least the activity they have at the beginning during the whole
;game
    (setq %upper-urgency% 600)
    (setq %first-urgency% 300)
    (setq %second-urgency% 150)
    (setq %third-urgency% 7)
    (setq %fourth-urgency% 4)
    (setq %fifth-urgency% 1)
    (create-coderack)
    (setq *name-counter* 1))

(defun quick ()
  (let (r)
    (setq r (cr-choose *coderack*))
    (print r) (terpri)
    (eval r)))

;REACTIVATE-CYTO FUNCTION
(defun reactivate-cyto ()
  (declare (special cyto-target cyto-brick1 cyto-brick2 cyto-brick3
                    cyto-brick4 cyto-brick5 *current-target*))
  (let (pnode)
    (setq pnode (car (send cyto-target :plinks)))
    (send (eval pnode) :add-activation %target-plus%)
    (setq pnode (car (send cyto-brick1 :plinks)))
    (send (eval pnode) :set-activation %brick-plus%)
    (setq pnode (car (send cyto-brick2 :plinks)))
    (send (eval pnode) :set-activation %brick-plus%)
    (setq pnode (car (send cyto-brick3 :plinks)))
    (send (eval pnode) :set-activation %brick-plus%)
    (setq pnode (car (send cyto-brick4 :plinks)))
    (send (eval pnode) :set-activation %brick-plus%)
    (setq pnode (car (send cyto-brick5 :plinks)))
```

```
    (send (eval pnode) :set-activation %brick-plus%)
    (setq pnode (car (send (eval (send *current-target* :name))
                           :plinks)))
    (if pnode (send (eval pnode) :add-activation %dtarget-plus%))
    (repump)))

;REFRESH-EVERYTHING FUNCTION
(defun refresh-everything ()
    (declare (special *iteration* *coderack* %third-urgency% %graphics% *pnet*))
;Loads new codelets on the coderack (random associations)
;Updates the interest of the blocks in the cytoplasm and
;checsk the temperature
;Updates the pnet by spreading the activation
;Loads codelets coming from the Pnet
  (let (nn)
    (cr-hang *coderack* '(look-for-new-block) %fourth-urgency%)
    (decrease-interest) (check-temperature)
    (spread-activation-in-pnet)
    (when %graphics%
       (update-pnet-display *pnet*)
       (setq nn (get-pname (concat 'gr *iteration*)))
       (dump-window  nn))
     (populate-coderack)))
```

```
(defvar *iteration* 0 "This is the iteration counter we use at top level")

(defun config (t1 b1 b2 b3 b4 b5)
    (initialize-pnet)
    (send result+ :set-instances '(("5g" "result+")))
    (send resultx :set-instances '(("5g" "resultx")))
    (send operand :set-instances '(("5g" "operand")))
    (send similar :set-instances '(("5g" "similar")))
    (send operation :set-instances '(("6g" "operation")))
    (send instance :set-instances '(("6g" "instance")))
    (send node-add :set-instances '(("6g" "node-add")))
    (send node-subtract :set-instances '(("6g" "node-subtract")))
    (send node-multiply :set-instances '(("6g" "node-multiply")))
    (init-cytoplasm t1 b1 b2 b3 b4 b5)
    (format t "Le jeu des chiffres") (terpri)
    (setq t1 (send *cytoplasm* :target)) (format t "Initial configuration :")
    (terpri) (format t "   Target : ~a" t1) (terpri)
    (setq b1 (send *cytoplasm* :brick1))
    (setq b2 (send *cytoplasm* :brick2))
    (setq b3 (send *cytoplasm* :brick3))
    (setq b4 (send *cytoplasm* :brick4))
    (setq b5 (send *cytoplasm* :brick5))
    (format t "   Bricks : ~a ~a ~a ~a ~a" b1 b2 b3 b4 b5) (terpri)
    (setq retry t)
;Graphics
   (cond
    (%graphics% (init-pnet-graphics *pnet*) (display-pnet *pnet*)))
;Initialization of the target and start of the outer loop
    (setq *problem-solved* 0)
    (cr-empty-coderack *coderack*)
    (read-target)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (setq %resultx% 200)
    (setq %result+% 100)
    (setq %operand% 50)
    (repump)
;Initialization of the bricks and first processing
    (read-brick 1)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (read-brick 2)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (read-brick 3)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (read-brick 4)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
    (read-brick 5)
    (eval (cr-choose *coderack*))
    (eval (cr-choose *coderack*))
;I start at x = 11 to have an early spreading of activation in the
;pnet (after 9 iterations).
```

```
    (setq x  11)
    (do ((y 0 (add1 y)))
     ((= 1 *problem-solved*) (format t "Done : ") (decompose 'cyto-target))
     (setq *iteration* y)
     (setq x (add1 x))
     (cond
         ((and (> x 400) (> (temperature) %temperature-threshold%))
          (cr-empty-coderack *coderack*) (setq x 39))
         ((= 0 (mod x 40))
         (refresh-everything) (reactivate-cyto))
         ((= 0 (mod x 20))
          (refresh-everything))
;          (terpri) (terpri) (print (temperature)) (terpri))
         ((= 0 (mod x 5))
        (cr-hang *coderack* '(look-for-new-block) %fourth-urgency%)
          (decrease-interest) (check-temperature))
         (t (setq rescod (if (cr-empty? *coderack*)
                             then nil
;                            else (cr-choose *coderack* t)))
                             else (cr-choose *coderack*)))
;If there is no more codelets, a last activation of the pnet is tried.
      (if (null rescod) then
          (cond
              (retry
                   (cr-hang *coderack* '(look-for-new-block)
               %fourth-urgency%)
                   (cr-hang *coderack* '(look-for-new-block)
               %fourth-urgency%)
                   (cr-hang *coderack* '(look-for-new-block)
               %fourth-urgency%)
                   (cr-hang *coderack* '(look-for-new-block)
               %fourth-urgency%)
                   (spread-activation-in-pnet)
                   (cond
                      (%graphics% (update-pnet-display *pnet*)
                                  (setq nn (get-pname (concat 'gr x)))
                                  (dump-window  nn)))
                   (setq retry nil)
                   (populate-coderack) (reactivate-cyto))
               (t (return))))
     (eval rescod)))))
  ;  (print rescod) (terpri)
   ;  (eval (car rescod))))))
```

host	doug
owner	defays (Daniel Defays)
printer	FARG Imagen
site	FARG World Headquarters
spooldate	Thu Jun 25 15:53:14 1987
files	pnet-graphics.l

```
language: printer
jamresistance: on
jobheader: on
pagereversal: on

IMAGEN Printing System, Version 2.1, Serial #84:12:136
Page images processed: 4
Pages printed: 4

Number of job messages: 2
Paper size (width, height):
  2560, 3328
Document length:
  8149 bytes
```