```
;*****************************************************************************************
;* A CALCULUS FOR MUSIC PERFORMANCE EXPRESSION                                           *
;* (c) 1991,  Henkjan Honing & Peter Desain                                              *
;*                                                                                       *
;* in CLOS (Common Lisp), uses loop macro                                                *
;*****************************************************************************************
;*****************************************************************************************
; musical objects
;*****************************************************************************************
;*****************************************************************************************
; abstract classes of musical objects

;(defpackage calculus)
;(in-package calculus)

(defclass musical-object ()
  ((name :reader name :initarg :name :initform 'no-name :type symbol)
   (score-onset :reader score-onset :type rational :initform 0)
   (left :reader left :initform nil)
   (right :reader right :initform nil))
  (:documentation "Musical Object"))

(defclass structured (musical-object)
  ((score-offset :reader score-offset :type rational))
  (:documentation "Structured Musical Object"))

(defclass multilateral (structured)
  ((components :reader components :initarg :components))
  (:documentation "Multilateral Musical Object"))

(defclass collateral (structured)
  ((main :reader main :initarg :main)
   (ornament :reader ornament :initarg :ornament))
  (:documentation "Ornamented Musical Object"))

(defclass successive (structured)
  ()
  (:documentation "Successive Musical Object"))

(defclass simultaneous (structured)
  ()
  (:documentation "Simultaneous Musical Object"))

(defclass basic (musical-object)
  ((score-offset :reader score-offset :type rational :initarg :score-dur))
  (:documentation "Basic Musical Object"))

;*****************************************************************************************
; instantiatable classes of musical objects

(defclass S (multilateral successive) () (:documentation "Sequential"))
(defclass P (multilateral simultaneous) () (:documentation "Parallel"))
(defclass ACCIA (collateral simultaneous) () (:documentation "Acciaccature"))
(defclass APPOG (collateral successive) () (:documentation "Appoggiature"))

(defclass NOTE (basic)
  ((dynamic :accessor dynamic :type float :initarg :dynamic)
   (perf-onset :accessor perf-onset :type float :initarg :perf-onset :initform nil)
   (perf-offset :accessor perf-offset :type float :initarg :perf-offset :initform nil))
  (:documentation "Note"))

(defclass PAUSE (basic) () (:documentation "Rest"))

;*****************************************************************************************
; creators for musical objects

(defun S (name &rest components)
  (make-instance 'S :name name :components components))

(defun P (name &rest components)
  (make-instance 'P :name name :components components))

(defun ACCIA (name ornament main)
 (make-instance 'ACCIA :name name :ornament ornament :main main))

(defun APPOG (name ornament main)
  (make-instance 'APPOG :name name :ornament ornament :main main))

(defun NOTE (&key name perf-onset perf-offset score-dur (dynamic 1))
  (make-instance 'NOTE :name name
                 :perf-onset perf-onset
                 :perf-offset perf-offset
                 :score-dur score-dur
                 :dynamic dynamic))
```

```
(defun PAUSE (&key name score-dur)
  (make-instance 'PAUSE :name name :score-dur score-dur))

;*****************************************************************************************
; extra acces functions for musical objects

(defmethod components ((object basic)) nil)
(defmethod components ((object collateral))
  (list (ornament object)(main object)))

(defmethod all-notes ((object musical-object))
  (loop for component in (components object) append (all-notes component)))

(defmethod all-notes ((object note)) (list object))

(defun has-name? (&rest names)
  #'(lambda (object &rest ignore)(member (name object) names)))

(defmethod find-parts ((object musical-object) pred)
  (if (funcall pred object)
    (list object)
    (loop for component in (components object)
          append (find-parts component pred))))

;*****************************************************************************************
; initialization of score times and context of musical objects

(defmethod initialize-instance :after ((object musical-object) &rest ignore)
  (object-check object)
  (initialize-score-times object)
  (initialize-context object))

(defmethod object-check ((object musical-object)) nil)

;*****************************************************************************************
; initialization of score-onset and offset of musical objects

(defmethod initialize-score-times ((object basic)))

(defmethod initialize-score-times ((object P))
  (setf (slot-value  object 'score-offset)
        (slot-value  (first (components object)) 'score-offset)))

(defmethod initialize-score-times ((object S))
  (loop with onset = 0
        for component in (components object)
        do (shift-score component onset)
        (setf onset (slot-value  component 'score-offset))
        finally (setf (slot-value  object 'score-offset) onset)))

(defmethod initialize-score-times ((object collateral))
 (setf (slot-value  object 'score-offset)
       (slot-value  (main object) 'score-offset)))

(defmethod initialize-score-times :after ((object APPOG))
  (shift-score (ornament object)
             (- (slot-value  (ornament object) 'score-offset))))

(defmethod shift-score  ((object musical-object) shift)
  (incf (slot-value object 'score-onset) shift)
  (incf (slot-value object 'score-offset) shift)
  (loop for component in (components object) do (shift-score component shift)))

;*****************************************************************************************
; initialization of context of musical objects

(defmethod initialize-context ((object musical-object)))

(defmethod initialize-context ((object S))
  (loop for component in (components object)
        for next-component in (rest (components object))
        do (set-contexts component next-component)))

(defmethod initialize-context ((object APPOG))
  (set-context (ornament object) (main object) 'right))

(defmethod set-contexts ((left musical-object) (right musical-object))
  (set-context left right 'right)
  (set-context right left 'left))

(defmethod set-context ((object musical-object) (context musical-object) dir)
  (setf (slot-value object dir) context))
```

```
(defmethod set-context :after ((object P) (context musical-object) dir)
  (loop for component in (components object)
        do (set-context component context dir)))

(defmethod set-context :after ((object S) (context musical-object) dir)
  (if (eql dir 'left)
    (set-context (first (components object)) context dir)
    (set-context (last-element (components object)) context dir)))

(defmethod set-context :after ((object collateral) (context musical-object) dir)
  (set-context (main object) context dir))

(defmethod set-context :after ((object ACCIA) (context musical-object) dir)
  (when (eql dir 'left)
    (set-context (ornament object) context dir)))

;*****************************************************************************************
;*****************************************************************************************
; maps
;*****************************************************************************************
;*****************************************************************************************
; abstract classes of maps

(defclass map ()
  ((sections :accessor sections :initarg :sections))
  (:documentation "Expression Map"))

(defclass multilateral-map (map)())
(defclass collateral-map (map)())
(defclass simultaneous-map (map)())
(defclass successive-map (map)())

;*****************************************************************************************
; instantiable classes of maps

(defclass P-map   (multilateral-map simultaneous-map)())
(defclass S-map   (multilateral-map successive-map)())
(defclass ACCIA-map (collateral-map simultaneous-map)())
(defclass APPOG-map (collateral-map successive-map)())

;*****************************************************************************************
; creator for maps

(defun make-map (sections)
  (let ((ordered-sections (sort sections #'< :key #'score-onset)))
    (cond ((null ordered-sections) nil)
          ((and (same-section-type? ordered-sections)
                (not-overlapping? ordered-sections))
           (make-instance (section-to-map (first ordered-sections))
                          :sections ordered-sections))
          (t (error "attempt to merge incompatible sections into expression map")))))

;*****************************************************************************************
; sections of maps
;*****************************************************************************************
; abstract classes of sections of maps

(defclass section ()
  ((all-score-times :accessor all-score-times :initarg :all-score-times)
   (all-expressions :accessor all-expressions :initarg :all-expressions))
  (:documentation "Expression Section"))

(defclass multilateral-section (section)())
(defclass collateral-section (section)())
(defclass successive-section (section)())
(defclass simultaneous-section (section)())

;*****************************************************************************************
; instantiable classes of sections of maps

(defclass S-section   (successive-section multilateral-section)())
(defclass P-section   (simultaneous-section multilateral-section)())
(defclass ACCIA-section (simultaneous-section collateral-section)())
(defclass APPOG-section (successive-section collateral-section)())

;*****************************************************************************************
; compatibility relation between musical objects, expression maps and sections thereof

(defmethod object-to-section ((object musical-object))
 (third (find (class-name (class-of object)) (object-network) :key #'first)))

(defmethod section-to-map ((section section))
  (second (find (class-name (class-of section)) (object-network) :key #'third)))
```

```
(defun object-network ()
  '((S S-map S-section)
    (P P-map P-section)
    (ACCIA ACCIA-map ACCIA-section)
    (APPOG APPOG-map APPOG-section)))

;***********************************************************************************************
; creators for sections of maps

(defun make-section (section-class all-score-times all-expressions)
  (make-instance section-class :all-score-times all-score-times :all-expressions all-expressions))

(defmethod make-new-section ((section section) expressions)
  (make-section (class-of section)
                (snoc (score-times section) (score-offset section))
                (snoc expressions (next-expression section))))

(defmethod make-new-section-from-pairs ((section section) pairs)
  (make-section (class-of section)
                (snoc (mapcar #'first pairs) (score-offset section))
                (snoc (mapcar #'second pairs) (next-expression section))))

;***********************************************************************************************
; extra accessors for sections of maps

(defmethod score-onset ((section section))
  (first (all-score-times section)))

(defmethod score-offset ((section section))
  (last-element (all-score-times section)))

(defmethod expressions ((section section))
  (butlast (all-expressions section)))

(defmethod next-expression ((section section))
  (last-element (all-expressions section)))

(defmethod score-times ((section section))
  (butlast (all-score-times section)))

(defmethod score-onset ((section collateral-section))
  (score-main section))

(defmethod main-expression ((section collateral-section))
  (second (all-expressions section)))

(defmethod ornament-expression ((section collateral-section))
  (first (all-expressions section)))

(defmethod score-main ((section collateral-section))
  (second (all-score-times section)))

(defmethod score-ornament ((section collateral-section))
  (first (all-score-times section)))

(defun same-section-type? (sections)
  (every #'(lambda (section) (class-of section)) sections))

(defun not-overlapping? (sections)
  (loop for section in sections
        for next-section in (rest sections)
        never (> (score-offset section) (score-onset next-section))))

;***********************************************************************************************
; find section (containing score time) in expression map

(defmethod lookup-section-containing ((map map) score-time)
  (loop for section in (sections map)
        when (<= (score-onset section) score-time (score-offset section))
        do (return section)))

;***********************************************************************************************
; lookup expression value (via score time) in expression map

(defmethod lookup-defined-expression ((map map) score-time)
  (lookup-defined-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-defined-expression (section score-time)
  (and section
       (loop for expression in (all-expressions section)
             for map-score-time in (all-score-times section)
             when (= map-score-time score-time)
             do (return expression))))
```

```
(defmethod lookup-expression ((map successive-map) score-time)
  (lookup-expression (lookup-section-containing map score-time) score-time))

(defmethod lookup-expression (section score)
  (and section
       (loop for expression in (all-expressions section)
             for expression-next in (rest (all-expressions section))
             for score-time in (all-score-times section)
             for score-time-next in (rest (all-score-times section))
             while (> score score-time-next)
             finally (return (interpolate score-time score score-time-next
                                          expression expression-next)))))

;*****************************************************************************************************
; lookup score time in a monotone rising expression map

(defmethod in-section-inverse? ((section section) expression)
  (and expression (<= (first (expressions section))
                      expression
                      (or (next-expression section)
                          (last-element (expressions section))))))

(defmethod lookup-inverse ((map S-map) expression)
  (loop for section in (sections map) thereis (lookup-inverse section expression)))

(defmethod lookup-inverse ((section section) expression)
  (and (in-section-inverse? section expression)
       (loop for expression-next in (rest (expressions section))
             for score-time in (score-times section)
             for score-time-next in (rest (score-times section))
             while (> expression expression-next)
             finally (return (list score-time score-time-next)))))

;*****************************************************************************************************
; mapping through expression maps, naamgeving !!

(defmethod map-map (fun (map map))
  (make-map (loop for section in (sections map) collect (funcall fun section))))

;*****************************************************************************************************
; mapping through filtered expression maps

(defmethod with-filtered-null-expression (fun (map map))
  (unfilter-null-expression (funcall fun (filter-null-expression map))
                            (filter-null-expression-out map)))

(defmethod filter-null-expression ((map map))
  (map-map #'filter-null-expression map))

(defmethod filter-null-expression ((section section))
  (make-new-section-from-pairs section
   (loop for expression in (expressions section)
         for score-time in (score-times section)
         when expression
         collect (list score-time expression))))

(defmethod filter-null-expression-out ((map map))
  (mapcar #'filter-null-expression-out (sections map)))

(defmethod filter-null-expression-out ((section section))
  (loop for expression in (expressions section)
        for score-time in (score-times section)
        for index from 0
        unless expression
        collect (list index score-time)))

(defmethod unfilter-null-expression ((map map) rejections)
  (make-map (mapcar #'unfilter-null-expression (sections map) rejections)))

(defmethod unfilter-null-expression ((section section) removed)
  (if removed
    (make-new-section-from-pairs section
                                 (loop with expressions = (expressions section)
                                       with score-times = (score-times section)
                                       for index from 0
                                       while (or score-times removed)
                                       when (and removed (= index (caar removed)))
                                       collect (list (second (pop removed)) nil)
                                       else collect (list (pop score-times) (pop expressions))))
    section))

;*****************************************************************************************************
;*****************************************************************************************************
; expression
```

```
;*********************************************************************************************
;*********************************************************************************************

(defclass expression ()())

;*********************************************************************************************
; nil and rests carry no expression, nil expressions and sections are not set


(defmethod get-expression      ((object null)(expression expression)) nil)
(defmethod get-next-expression ((object null)(expression expression)) nil)

(defmethod get-expression ((object PAUSE)(expression expression)) nil)
(defmethod set-expression ((object PAUSE)(expression expression) value) nil)

(defmethod set-expression ((object musical-object) expression value-or-section) nil)
(defmethod get-next-expression ((object musical-object)(expression expression))
  (get-expression (right object) expression))

;*********************************************************************************************
; get expression of notes

(defmethod get-notes-expression ((object musical-object) (expression expression))
  (loop for note in (all-notes object)
        collect (fetch-expression note expression)))

(defmethod set-notes-expression ((object musical-object) (expression expression) values )
  (loop for note in (all-notes object)
        for value in values
        do (set-expression note expression value)))

;*********************************************************************************************
; propagate expression (interpolated, truncating-shift and shift)

(defmethod propagate-interpolated ((object S) old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
        do (propagate-interpolated component old-begin new-begin old-end new-end expression)))

(defmethod propagate-interpolated ((object P) old-begin new-begin old-end new-end expression)
  (loop for component in (components object)
        do (propagate-truncating-shift component (save-- new-begin old-begin) new-end expression)))

(defmethod propagate-interpolated ((object collateral) old-begin new-begin old-end new-end expression)
  (let* ((ref (fetch-expression (main object) expression))
         (shift (save-- (interpolate old-begin ref old-end new-begin new-end) ref)))
    (propagate-interpolated (main object) old-begin new-begin old-end new-end expression)
    (propagate-shift (ornament object) shift expression)))

(defmethod propagate-interpolated ((object NOTE) old-begin new-begin old-end new-end expression)
  (set-expression
   object expression
   (interpolate old-begin (fetch-expression object expression) old-end new-begin new-end)))

(defmethod propagate-interpolated ((object PAUSE) old-begin new-begin old-end new-end expression))

;*********************************************************************************************
; propagate-truncating-shift

(defmethod propagate-truncating-shift :around ((object musical-object) shift end expression)
  (when shift (call-next-method)))

(defmethod propagate-truncating-shift ((object multilateral) shift end expression)
  (loop for component in (components object)
        do (propagate-truncating-shift component shift end expression)))

(defmethod propagate-truncating-shift ((object collateral) shift end expression)
  (propagate-shift (ornament object) shift expression)
  (propagate-truncating-shift (main object) shift end expression))

(defmethod propagate-truncating-shift ((object NOTE) shift end expression)
  (set-expression object
                  expression
                  (save-min (save-+ (fetch-expression object expression) shift) end)))

(defmethod propagate-truncating-shift ((object PAUSE) shift end expression))

;*********************************************************************************************
; propagate-shift

(defmethod propagate-shift :around ((object musical-object) shift expression)
  (when shift (call-next-method)))

(defmethod propagate-shift ((object structured) shift expression)
  (loop for component in (components object)
```

```
        do (propagate-shift component shift expression)))

(defmethod propagate-shift ((object basic) shift expression)
  (set-expression object
                  expression
                  (save-+ (fetch-expression object expression) shift)))

;****************************************************************************************************
; onset timing
;****************************************************************************************************

(defclass expressive-timing (expression)  ())
(defclass onset-timing (expressive-timing)  ())
(defclass basic-asynchrony (onset-timing) ())
(defclass basic-tempo (onset-timing) ())

(defclass estimate-onset-timing (onset-timing estimate-mixin) ()) ;for use-timing in articulation

;****************************************************************************************************
; get expressive timing

(defmethod get-expression ((object NOTE) (expression onset-timing))
  (perf-onset object))

(defmethod get-expression ((object S) (expression onset-timing))
   (get-expression (first (components object)) expression))

(defmethod get-expression ((object P) (expression onset-timing))
  (loop for component in (components object)
        when (get-expression component expression)
        minimize it))

(defmethod get-expression ((object collateral) (expression onset-timing))
  (get-expression (main object) expression))

;****************************************************************************************************
; set expressive timing

(defmethod set-expression ((object NOTE) (expression onset-timing) value)
  (setf (perf-onset object) value))

(defmethod set-expression ((object S) (expression onset-timing) (section S-section))
  (loop for new-expression in (expressions section)
        for next-new-expression in (snoc (rest (expressions section)) (next-expression section))
        for component in (components object)
        do (propagate-interpolated component
                                   (fetch-expression component expression)
                                   new-expression
                                   (fetch-expression (right component) expression)
                                   next-new-expression
                                   expression)))

(defmethod set-expression ((object P) (expression onset-timing) (section P-section))
  (loop for new-expression in (expressions section)
        for component in (components object)
        do (propagate-truncating-shift component
                                       (save-- new-expression
                                               (fetch-expression component expression))
                                       (get-next-expression object expression)
                                       expression)))

(defmethod set-expression ((object ACCIA) (expression onset-timing) (section ACCIA-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                           (fetch-expression (ornament object) expression))
                   expression))

(defmethod set-expression ((object APPOG) (expression onset-timing) (section APPOG-section))
  (propagate-interpolated (ornament object)
                          (fetch-expression (ornament object) expression)
                          (ornament-expression section)
                          (fetch-expression (right (ornament object)) expression)
                          (main-expression section)
                          expression))

;****************************************************************************************************
; scale expressive-timing

(defmethod scale-expression ((section P-section)
                             (expression basic-asynchrony)
                             factor)
  (if (expressions section)
    (make-new-section
     section
```

```
     (scale-P-expression-points (expressions section) factor))
    section))

(defmethod scale-expression ((section S-section)
                             (expression basic-tempo)
                             factor)
  (cond ((and (expressions section)(next-expression section))
         (scale-S-section-] section factor))
        ((rest (expressions section))
         (scale-S-section-> section factor))
        (t section)))

(defmethod scale-S-section-] ((section section) factor)
    (make-new-section section  (scale-S-expression-points
                                (snoc (score-times section)(score-offset section))
                                (snoc (expressions section) (next-expression section))
                                factor)))

(defmethod scale-S-section-> ((section section) factor)
  (make-new-section section
                    (scale-S-expression-points (score-times section)
                                               (expressions section)
                                               factor)))

(defmethod scale-expression ((section ACCIA-section) (expression basic-asynchrony) factor)
  (make-new-section section
                      (scale-ACCIA-points (main-expression section)
                                          (ornament-expression section)
                                          factor)))

(defmethod scale-expression ((section APPOG-section) (expression basic-tempo) factor)
  (make-new-section section
                    (scale-APPOG-points (ornament-expression section)
                                        (main-expression section)
                                        (next-expression section)
                                        (score-ornament section)
                                        (score-main section)
                                        (score-offset section)
                                        factor)))

;*****************************************************************************************

(defun scale-P-expression-points (perf-onsets factor)
  (let* ((perf-begin (apply #'min perf-onsets))
         (perf-iois (mapcar #'(lambda (onset) (- onset perf-begin)) perf-onsets))
         (raw-new-perf-iois (mapcar #'(lambda (perf)(scale-expression-lin perf factor))
                                    perf-iois))
         (shift (- (apply #'min raw-new-perf-iois)))
         (new-perf-onsets (mapcar #'(lambda (ioi) (+ ioi shift perf-begin)) raw-new-perf-iois)))
    new-perf-onsets))

(defun scale-S-expression-points (score-times perf-times factor)
  (let* ((perf-iois (mapcar #'- (rest perf-times) perf-times))
         (score-iois (mapcar #'- (rest score-times) score-times))
         (perf-begin (first perf-times))
         (perf-end (last-element perf-times))
         (raw-new-perf-iois (mapcar #'(lambda (score perf)
                                        (scale-velocity score perf factor))
                                    score-iois
                                    perf-iois))
         (new-perf-iois (normalise raw-new-perf-iois (- perf-end perf-begin)))
         (new-perf-times (integrate new-perf-iois perf-begin)))
    new-perf-times))

(defun scale-ACCIA-points (main-expression ornament-expression factor)
  (let* ((expression-interval (- main-expression ornament-expression))
         (new-expression-ornament (- main-expression
                                     (scale-expression-lin expression-interval factor))))
    (list new-expression-ornament main-expression)))

(defun scale-APPOG-points  (ornament-expression main-expression next-expression
                                score-ornament score-main score-end
                                   factor)
  (let* ((score-ornament-ioi (- score-main score-ornament ))
         (expression-ornament-ioi (- main-expression ornament-expression))
         (score-main-ioi (- score-end score-main))
         (expression-main-ioi (- next-expression main-expression))
         (ornament-tempo (/ score-ornament-ioi expression-ornament-ioi))
         (main-tempo (/ score-main-ioi expression-main-ioi))
         (relative-tempo (/ ornament-tempo main-tempo))
         (new-ornament-tempo (* main-tempo (expt relative-tempo factor)))
         (new-expression-ornament-ioi (/ score-ornament-ioi new-ornament-tempo))
         (new-expression-ornament (- main-expression new-expression-ornament-ioi)))
    (list new-expression-ornament main-expression next-expression)))
```

```
;*********************************************************************************************
; expression scale methods

(defun scale-velocity (score perf factor)
  "Exponential scaling"
  (/ score (expt (/ score perf) factor)
     ))

(defun scale-expression-lin  (perf factor)
  "Linear scaling"
  (* perf factor))

;*********************************************************************************************
; stretch expressive-timing

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression onset-timing))
  (make-new-section
   section
   (loop for perf-time in (expressions section)
         as (score-begin score-end) = (lookup-inverse old perf-time)
         collect (if (and score-begin score-end)
                     (interpolate (lookup-expression old score-begin)
                                  perf-time
                                  (lookup-expression old score-end)
                                  (lookup-expression new score-begin)
                                  (lookup-expression new score-end))
                     perf-time))))

;*********************************************************************************************
; mixin to estimate expression in case of absence, by linear inter- or extrapolation
;*********************************************************************************************

(defclass estimate-mixin () ())

(defmethod fetch-expression :around ((object musical-object) (expression estimate-mixin))
  (or (get-expression object expression)
      (estimate-expression object expression)))

(defmethod fetch-expression ((object null) (expression expression)) nil)

(defmethod fetch-expression ((object musical-object) (expression expression))
  (get-expression object expression))

(defmethod get-next-expression :around ((object musical-object) (expression estimate-mixin))
  (cond ((call-next-method))
        ((right object)
         (estimate-expression (right object) expression))
        (t
         (estimate-next-expression object expression))))

(defmethod fetch-onset :around ((object musical-object) (expression estimate-mixin))
  (fetch-expression object (find-expression 'estimate-onset-timing)))

(defmethod estimate-expression ((object musical-object) (expression expression))
  (estimate-context (context-with-expression object expression #'left)
                    object
                    (context-with-expression object expression #'right)
                    expression
                    t))

(defmethod estimate-next-expression ((object musical-object) (expression expression))
  (let* ((left (context-with-expression object expression #'left))
         (lefter (and left
                      (left left)
                      (context-with-expression (left left) expression #'left))))
    (when (and left lefter)
      (interpolate (score-onset lefter)
                   (score-offset object)
                   (score-onset left)
                   (get-expression lefter expression)
                   (get-expression left expression)))))

(defmethod estimate-context (left object right (expression expression) first-try)
  (cond ((and left right)
         (interpolate (score-onset left)
                      (score-onset object)
                      (score-onset right)
                      (get-expression left expression)
                      (get-expression right expression)))
        ((and left (left left) first-try)
```

```
         (estimate-context (context-with-expression (left left) expression #'left)
                           object
                           left
                           expression nil))
        ((and right (right right) first-try)
         (estimate-context right
                           object
                           (context-with-expression (right right) expression #'right)
                           expression nil))
        (t nil)))

(defmethod context-with-expression ((object musical-object) (expression expression) direction)
  (cond ((get-expression object expression)
         object)
        ((funcall direction object)
         (context-with-expression (funcall direction object) expression direction))
        (t nil)))

;*************************************************************************************************
; keeping articulation invariant: mixin for expressive timing expression
;*************************************************************************************************

(defclass keep-articulation-mixin () ())
(defclass keep-overlap-articulation-mixin (keep-articulation-mixin)())
(defclass keep-duration-articulation-mixin (keep-articulation-mixin)())
(defclass keep-proportion-articulation-mixin (keep-articulation-mixin)())

(defmethod articulation ((expression keep-overlap-articulation-mixin))
  (find-expression 'basic-overlap-articulation))

(defmethod articulation ((expression keep-duration-articulation-mixin))
  (find-expression 'basic-duration-articulation))

(defmethod articulation ((expression keep-proportion-articulation-mixin))
  (find-expression 'basic-proportion-articulation))

(defmethod set-map :around ((object musical-object) map (expression keep-articulation-mixin) ground)
  (when map
    (let* ((parts (find-parts object ground))
           (articulation-collections
            (loop for part in parts collect (get-notes-expression part (articulation expression)))))
      (call-next-method)
      (loop for part in parts
            for collection in articulation-collections
            do (set-notes-expression part (articulation expression) collection))))
    object)

;*************************************************************************************************
; resource for expression instances

(defvar *expression-instances*)
(setf *expression-instances* nil)
(defvar *use-expression-resource*)
(setf *use-expression-resource* t)

(defun find-expression (class)
  (or (and *use-expression-resource*
           (cdr (assoc class *expression-instances*)))
      (make-expression-instance class)))

(defun make-expression-instance (class)
  (let ((instance (make-instance class)))
    (when *use-expression-resource*
      (push (cons class instance) *expression-instances*))
    instance))

;*************************************************************************************************
; averaging expression
;*************************************************************************************************

(defclass averaging-expression-mixin ()()) ; waarom mixin ??

;*************************************************************************************************
; get averaging expression

(defmethod get-expression ((object multilateral) (expression averaging-expression-mixin))
  (loop for component in (components object)
        when (get-expression component expression)
        sum it into total
        finally (return (/ total (length (components object))))))

(defmethod get-expression ((object collateral) (expression averaging-expression-mixin))
  (get-expression (main object) expression))
```

```
;*********************************************************************************************
; set averaging expression

(defmethod set-expression ((object multilateral) (expression averaging-expression-mixin)
                           (section multilateral-section))
  (loop for component in (components object)
        for new-expression in (expressions section)
        do (propagate-shift component
                            (save-- new-expression
                                    (fetch-expression component expression))
                            expression)))

(defmethod set-expression ((object collateral)
                           (expression averaging-expression-mixin)
                           (section collateral-section))
  (propagate-shift (ornament object)
                   (save-- (ornament-expression section)
                                        (fetch-expression (ornament object) expression))
                   expression))

;*********************************************************************************************
; scale averaging expression

(defmethod scale-expression ((section multilateral-section)
                             (expression averaging-expression-mixin)
                             factor)
   (let* ((mean-expression (mean (expressions section)))
          (expression-deviations (mapcar #'(lambda(expression)
                                             (- expression mean-expression))
                                         (expressions section)))
          (new-expressions (mapcar #'(lambda (expression-deviation)
                                               (+ mean-expression
                                                  (scale-expression-lin expression-deviation factor)))
                                           expression-deviations)))
     (make-new-section section new-expressions)))

(defmethod scale-expression ((section collateral-section)
                             (expression averaging-expression-mixin)
                             factor)
  (let* ((expression-deviation (- (ornament-expression section)
                                  (main-expression section)))
         (new-ornament-expression  (+ (main-expression section)
                                      (scale-expression-lin expression-deviation factor))))
    (make-new-section section
                      (list new-ornament-expression
                            (main-expression section)))))

;*********************************************************************************************
; stretch averaging expression

(defmethod stretch-expression ((section S-section)
                               (old S-map)
                               (new S-map)
                               (expression averaging-expression-mixin))
  (make-new-section
   section
   (loop for expression in (expressions section)
         for score-time in (score-times section)
         as old-expression = (lookup-expression old score-time)
         as new-expression = (lookup-expression new score-time)
         as stretched-expression = (if (and old-expression new-expression expression)
                                       (+ expression (- new-expression old-expression))
                                       expression)
         collect stretched-expression)))

;*********************************************************************************************
; articulation
;*********************************************************************************************

(defclass offset-timing (expressive-timing) ())
(defclass articulation (offset-timing averaging-expression-mixin)())
(defclass basic-overlap-articulation (articulation)())
(defclass basic-duration-articulation (articulation)())
(defclass basic-proportion-articulation (articulation)())

;*********************************************************************************************

(defmethod get-expression ((object NOTE) (expression offset-timing))
  (perf-offset object))

(defmethod fetch-onset ((object musical-object) (expression articulation))
  (get-expression object (find-expression 'onset-timing)))

;*********************************************************************************************
```

```
; get articulation

#|
(defmethod get-expression :around ((object NOTE) (expression basic-overlap-articulation))
   (when (right object)
    (save-- (call-next-method)
            (fetch-onset (right object) expression))))

(defmethod get-expression :around ((object NOTE) (expression basic-duration-articulation))
  (- (call-next-method)
     (fetch-onset object expression)))

(defmethod get-expression :around ((object NOTE) (expression basic-proportion-articulation))
  (when (and (fetch-onset object expression)
             (right object)
             (fetch-onset (right object) expression))
    (/ (- (call-next-method)
          (fetch-onset object expression))
       (- (fetch-onset (right object) expression)
          (fetch-onset object expression)))))

|#

(defmethod get-expression  ((object NOTE) (expression basic-overlap-articulation))
   (when (right object)
    (save-- (perf-offset object)
            (fetch-onset (right object) expression))))

(defmethod get-expression ((object NOTE) (expression basic-duration-articulation))
  (- (perf-offset object)
     (fetch-onset object expression)))

(defmethod get-expression ((object NOTE) (expression basic-proportion-articulation))
  (when (and (fetch-onset object expression)
             (right object)
             (fetch-onset (right object) expression))
    (/ (- (perf-offset object)
          (fetch-onset object expression))
       (- (fetch-onset (right object) expression)
          (fetch-onset object expression)))))

;*****************************************************************************************************
; set articulation

(defmethod set-expression ((object NOTE) (expression basic-overlap-articulation) value)
  (when (and (right object) (fetch-onset (right object) expression))
    (setf (perf-offset object)
          (max (fetch-onset object expression)
               (+ (fetch-onset (right object) expression)
                  value)))))

(defmethod set-expression ((object NOTE) (expression basic-duration-articulation) value)
  (setf (perf-offset object)
        (+ (fetch-onset object expression)
           (max 0 value))))

(defmethod set-expression ((object NOTE) (expression basic-proportion-articulation) value)
  (when (and (right object)(perf-onset (right object)))
    (setf (perf-offset object)
          (+ (fetch-onset object expression)
             (* (- (fetch-onset (right object) expression)
                   (fetch-onset object expression))
                (max 0 value))))))

;*****************************************************************************************************
; empty expression (to recover only score times)
;*****************************************************************************************************

(defclass empty-expression (expression)  ())

(defmethod get-expression ((object musical-object) (expression empty-expression)) nil)

;*****************************************************************************************************
; mixing instantiable classes of expression
;*****************************************************************************************************

(defmacro class-mixer (&rest class-cocktail-pairs)
  (list* 'prog1 t
         (loop for tuples on class-cocktail-pairs by #'cdddr
               as name = (first tuples)
               as doc = (second tuples)
               as cocktail = (third tuples)
               collect `(defclass ,name ,cocktail ()
                          (:documentation ,doc)))))
```

```
(class-mixer
  tempo " "
  (basic-tempo)

  asynchrony " "
  (basic-asynchrony)

  estimate-tempo " "
  (basic-tempo estimate-mixin)

  estimate-asynchrony " "
  (basic-asynchrony estimate-mixin)

  keep-overlap-articulation-tempo " "
  (basic-tempo keep-overlap-articulation-mixin)

  keep-duration-articulation-tempo " "
  (basic-tempo keep-duration-articulation-mixin)

  keep-proportion-articulation-tempo " "
  (basic-tempo keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-tempo " "
  (basic-tempo keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-tempo " "
  (basic-tempo keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-tempo " "
  (basic-tempo keep-proportion-articulation-mixin estimate-mixin)

  keep-overlap-articulation-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin)

  keep-duration-articulation-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin)

  keep-proportion-articulation-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin)

  keep-overlap-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-overlap-articulation-mixin estimate-mixin)

  keep-duration-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-duration-articulation-mixin estimate-mixin)

  keep-proportion-articulation-estimate-asynchrony " "
  (basic-asynchrony keep-proportion-articulation-mixin estimate-mixin)

  overlap-articulation " "
  (basic-overlap-articulation)

  duration-articulation " "
  (basic-duration-articulation)

  proportion-articulation " "
  (basic-proportion-articulation)

  estimate-overlap-articulation " "
  (basic-overlap-articulation estimate-mixin)

  estimate-duration-articulation " "
  (basic-duration-articulation estimate-mixin)

  estimate-proportion-articulation " "
  (basic-proportion-articulation estimate-mixin))

;****************************************************************************************************
;****************************************************************************************************
; extracting and imposing expression maps of musical objects using expression
;****************************************************************************************************
;****************************************************************************************************
; extracting a expression map

(defmethod get-map ((object musical-object) expression ground)
  (make-map (loop for part in (find-parts object ground)
                  collect (get-section part expression))))

(defmethod get-section ((object musical-object) expression)
  (make-section (object-to-section object)
                (snoc (mapcar #'score-onset (components object))
                      (score-offset object))
                (snoc (mapcar #'(lambda (component)
```

```
                                (fetch-expression component expression))
                             (components object))
                      (get-next-expression object expression))))

;*****************************************************************************************
; impose a expression map

(defmethod set-map ((object musical-object) map expression ground)
  (loop for part in (find-parts object ground)
        for section in (sections map)
        do (set-expression part expression section))
  object)

;*****************************************************************************************
;*****************************************************************************************
; operations on expression maps
;*****************************************************************************************
;*****************************************************************************************
; scale expression map

(defmethod scale-map ((map map) expression factor)
  (with-filtered-null-expression #'(lambda (filtered-map)
                           (scale-filtered-map filtered-map expression factor)) ;??
                       map))

(defmethod scale-filtered-map ((map map) expression factor)
  (map-map #'(lambda (section)
               (scale-expression section expression (get-parameter factor (score-onset section))))
           map))

;*****************************************************************************************
; interpolate S-expression maps

(defmethod interpolate-maps ((map1 S-map) (map2  S-map) factor)
  (map-map  #'(lambda (section) (interpolate-section section
                                                     (filter-null-expression map2)
                                                     factor))

                       map1))

(defmethod interpolate-section ((section S-section)(map S-map) factor)
  (make-new-section
   section
   (loop for score-time in (score-times section)
         for expression in (expressions section)
         collect (in-between expression
                             (lookup-expression map score-time)
                             (get-parameter factor score-time)))))

(defmethod monotonise-map ((map S-map))
  (map-map #'monotonise-section map))

(defmethod monotonise-section ((section S-section))
  (make-new-section
   section
   (loop for expression in (expressions section)
         when expression
         maximize expression into state
         and collect  state
         else collect nil)))

;*****************************************************************************************
; get S-expression maps at sync points

(defmethod get-sync-map ((map1 S-map) (map2  S-map))
  (map-map  #'(lambda (section) (get-sync-section section map2)) map1))

(defmethod get-sync-section ((section S-section) (map S-map))
  (make-new-section-from-pairs  section
   (loop for score-time in (all-score-times section)
         for expression in (all-expressions section)
         as new-expression = (and expression
                                  (lookup-defined-expression map score-time))
         when new-expression collect (list score-time expression))))

;*****************************************************************************************
; stretch expression map

(defmethod stretch-map ((map successive-map) (old successive-map) (new successive-map) expression)
  (let ((filtered-map (filter-null-expression map))
        (filtered-old (filter-null-expression old))
        (filtered-new (filter-null-expression new))
        (removed (filter-null-expression-out map)))
    (unfilter-null-expression
```

```
     (map-map
      #'(lambda (section)
          (stretch-expression section filtered-old filtered-new expression))
      filtered-map)
     removed)))

;****************************************************************************************************
;****************************************************************************************************
; time-changing parameters
;****************************************************************************************************
;****************************************************************************************************

(defun get-parameter (factor score-time)
  (if (numberp factor)
    factor
    (funcall factor score-time)))

(defun make-ramp (x1 x2 y1 y2) ; as s-section ??
  #'(lambda (x) (interpolate x1 x x2 y1 y2)))

;****************************************************************************************************
;****************************************************************************************************
; transformations on musical objects
;****************************************************************************************************
;****************************************************************************************************
; transfer expression transformation

(defmethod transfer ((object musical-object) expression foreground background)
  (let* ((foreground-map (get-map object expression foreground))
         (background-map (get-map object (find-expression 'empty-expression) background))
         (new-background-map (interpolate-maps background-map foreground-map 1)))
    (set-map object new-background-map expression background))
  object)

;****************************************************************************************************
; scale expression transformation

(defmethod scale ((object musical-object) expression foreground background factor)
  (let* ((old-foreground-map (get-map object expression foreground))
         (new-foreground-map (when old-foreground-map
                               (scale-map old-foreground-map expression factor)))
         (old-background-map (when background
                               (get-map object expression background)))
         (new-background-map (when old-background-map
                               (stretch-map old-background-map
                                            old-foreground-map
                                            new-foreground-map
                                            expression))))
    (when new-foreground-map
      (set-map object new-foreground-map expression foreground))
    (when new-background-map
      (set-map object new-background-map expression background)))
  object)

;****************************************************************************************************
; scale intervoice expression transformation

(defmethod scale-intervoice ((object musical-object) expression
                             voice1 voice2 factor ref)
  (let* ((map1 (get-map object expression voice1))
         (map2 (get-map object expression voice2)))
    (when (and map1 map2)
      (let* ((original-sync-map1 (get-sync-map map1 map2))
             (original-sync-map2 (get-sync-map map2 map1))
             (new-sync-map1 (monotonise-map (interpolate-maps
                                             original-sync-map1
                                             original-sync-map2 (* ref (- 1 factor)))))
             (new-sync-map2 (monotonise-map (interpolate-maps
                                             original-sync-map2
                                             original-sync-map1 (* (- 1 ref) (- 1 factor)))))
             (new-map1 (stretch-map
                        map1 original-sync-map1 new-sync-map1 expression))
             (new-map2 (stretch-map
                        map2 original-sync-map2 new-sync-map2 expression)))
        (set-map object new-map1 expression voice1)
        (set-map object new-map2 expression voice2)))
    object))

;****************************************************************************************************
;****************************************************************************************************
; lisp utilities
;****************************************************************************************************
;****************************************************************************************************
```

```
(defun last-element (list)
  (first (last list)))

(defun snoc (list item)
  (append list (list item)))

(defun mean (numbers)
  (/ (apply #'+ numbers) (length numbers)))

(defun save-min (&rest list)
  (let  ((new-list (remove nil list)))
    (and new-list (apply #'min new-list))))

(defun save-max (&rest list)
  (let  ((new-list (remove nil list)))
    (and new-list (apply #'max new-list))))

(defun save-- (&rest list)
  (and (notany #'null list)
       (apply #'- list)))

(defun save-+ (&rest list)
  (apply #'+ (remove nil list)))

(defun enforce-limits (minimum x maximum)
  (max minimum (min x maximum)))

(defun integrate (list start)
  (if (null list)
    (list start)
    (cons start
          (integrate (rest list) (+ (first list) start)))))

(defun normalise (list dur)
  (let ((factor  (/ dur (apply #'+ list))))
    (mapcar #'(lambda(item)(* factor item)) list)))

(defun interpolate (x1 x x2 y1 y2)
  (cond ((eql y1 y2) y1)
        ((eql x1 x2) nil)
        ((null x) nil)
        ((and x1 (= x x1)) y1)
        ((and x2 (= x x2)) y2)
        ((and x1 x2)
         (in-between y1 y2 (/ (- x x1) (- x2 x1))))
        (t nil)))

(defun in-between (y1 y2 a)
 (cond ((= a 0) y1)
       ((= a 1) y2)
       ((and y1 y2)
        (+ y1 (* a (- y2 y1))))
       (t nil)))


;*****************************************************************************************************
;*****************************************************************************************************
; examples
;*****************************************************************************************************
;*****************************************************************************************************

#|

(defun metre-example ()
  (S 'bars
     (P 'bar
        (S 'melody
           (PAUSE :name 'pause :score-dur 1/4)
           (NOTE :name 64 :score-dur 1/8
                 :perf-onset .30 :perf-offset 0.5 :dynamic .7))
        (S 'accompagniment
           (PAUSE :name 'pause :score-dur 3/8)))
     (P 'bar
        (S 'melody
           (APPOG 'appogiatura
                  (NOTE :name 64 :score-dur 1/8
                        :perf-onset .550 :perf-offset .680 :dynamic .75)
                  (NOTE :name 55  :score-dur 1/4
                        :perf-onset .675 :perf-offset 1.133 :dynamic .7))
           (NOTE :name 55 :score-dur 1/8
                 :perf-onset 1.125  :perf-offset 1.475 :dynamic .7))
        (S 'accompagniment
           (NOTE :name 38 :score-dur 1/8
                 :perf-onset .725 :perf-offset .90 :dynamic .6)
```

```
         (NOTE :name 43 :score-dur 1/8
               :perf-onset .95 :perf-offset 1.2 :dynamic .6)
         (NOTE :name 47 :score-dur 1/8
               :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)))
   (P 'bar
      (S 'melody
         (ACCIA 'acciaccatura
                (NOTE :name 59 :score-dur 1/16
                      :perf-onset 1.600 :perf-offset 1.7 :dynamic .65)
                (NOTE :name 57 :score-dur 1/8
                      :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
         (NOTE :name 55 :score-dur 1/8
               :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
         (NOTE :name 57 :score-dur 1/8
               :perf-onset 2.256 :perf-offset 2.647 :dynamic .65))
      (S 'accompagniment
         (P 'chord
            (NOTE :name 38 :score-dur 3/8
                  :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
            (NOTE :name 42 :score-dur 3/8
                  :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
            (NOTE :name 48 :score-dur 3/8
                  :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))))
   (P 'bar
      (S 'melody
         (NOTE :name 55 :score-dur 3/8
               :perf-onset 2.425 :perf-offset 4 :dynamic .7))
      (S 'accompagniment
         (P 'chord
            (NOTE :name 43 :score-dur 3/8
                  :perf-onset 2.500 :perf-offset 4 :dynamic .6)
            (NOTE :name 47 :score-dur 3/8
                  :perf-onset 2.550 :perf-offset 4 :dynamic .7)
            (NOTE :name 50 :score-dur 3/8
                  :perf-onset 2.580 :perf-offset 4.5 :dynamic .65))))))


(defun background-example ()
 (P 'fragment
   (S 'melody
      (PAUSE :name 'pause :score-dur 1/4)
      (NOTE :name 64 :score-dur 1/8
            :perf-onset 0.3 :perf-offset 0.5 :dynamic .7)
      (APPOG 'appogiatura
             (NOTE :name 64 :score-dur 1/8
                   :perf-onset .550 :perf-offset .680 :dynamic .75)
             (NOTE :name 55  :score-dur 1/4
                   :perf-onset .675 :perf-offset 1.133 :dynamic .7))
      (NOTE :name 55 :score-dur 1/8
            :perf-onset 1.125  :perf-offset 1.475 :dynamic .7)
      (ACCIA 'acciaccatura
             (NOTE :name 59 :score-dur 1/16
                   :perf-onset 1.600 :perf-offset 1.700 :dynamic .65)
             (NOTE :name 57 :score-dur 1/8
                   :perf-onset 1.625 :perf-offset 1.880 :dynamic .7))
      (NOTE :name 55 :score-dur 1/8
            :perf-onset 1.880 :perf-offset 2.256 :dynamic .6)
      (NOTE :name 57 :score-dur 1/8
            :perf-onset 2.256 :perf-offset 2.647 :dynamic .65)
      (NOTE :name 55 :score-dur 3/8
            :perf-onset 2.425 :perf-offset 4 :dynamic .7))
   (S 'accompagniment
      (PAUSE :name 'pause :score-dur 3/8)
      (NOTE :name 38 :score-dur 1/8
            :perf-onset .725 :perf-offset .90 :dynamic .6)
      (NOTE :name 43 :score-dur 1/8
            :perf-onset .950 :perf-offset 1.2 :dynamic .6)
      (NOTE :name 47 :score-dur 1/8
            :perf-onset 1.150 :perf-offset 1.475 :dynamic .7)
      (P 'chord
         (NOTE :name 38 :score-dur 3/8
               :perf-onset 1.725 :perf-offset 2.500 :dynamic .7)
         (NOTE :name 42 :score-dur 3/8
               :perf-onset 1.775 :perf-offset 2.500 :dynamic .65)
         (NOTE :name 48 :score-dur 3/8
               :perf-onset 1.800 :perf-offset 2.500 :dynamic .7))
      (P 'chord
         (NOTE :name 43 :score-dur 3/8
               :perf-onset 2.500 :perf-offset 4 :dynamic .6)
         (NOTE :name 47 :score-dur 3/8
               :perf-onset 2.550 :perf-offset 4 :dynamic .7)
         (NOTE :name 50 :score-dur 3/8
               :perf-onset 2.580 :perf-offset 4.5 :dynamic .65)))))
```

```
(scale (metre-example)
       (find-expression 'tempo)
       (has-name? 'bars)
       nil
       2)

(scale (metre-example)
       (find-expression 'asynchrony)
       (has-name? 'bar)
       nil
       2)

(scale (background-example)
       (find-expression 'tempo)
       (has-name? 'melody)
       nil
       2)

(scale (background-example)
       (find-expression 'tempo)
       (has-name? 'melody)
       (has-name? 'accompagniment)
       2)


|#
```