```
(defsystem ascparser
  :source-pathname "."
  :source-extension "lisp"
  :binary-pathname "."
  :binary-extension nil
  :components ((:module ascparser
                        :source-pathname "ascparser"
                        :binary-pathname "bin"
                        :components ((:file "ASCParser"
                                            :depends-on ("IOStream" "Object"))
                                     (:file "Object"
                                            :depends-on ("Math3D"))
                                     (:file "IOStream"
                                            :depends-on ("Tools"))
                                     (:file "Math3D"
                                            :depends-on ("Tools"))
                                     (:file "Tools"))))



  :depends-on nil)
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: IOStream.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:58 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------


(declaim (optimize (speed 0)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 3)))


;;;----------------------------------------------------------------------------
;;; Tool functions from Paul Graham
;;;----------------------------------------------------------------------------


;;Paul Graham's 'Ansi Common Lisp'
(defun tokens (str test start)
  (let ((p1 (position-if test str :start start)))
    (if p1
        (let ((p2 (position-if #'(lambda (c)
                                   (not (funcall test c)))
                               str :start p1)))
          (cons (subseq str p1 p2)
                (if p2
                    (tokens str test p2)
                  nil)))
      nil)))


;;;----------------------------------------------------------------------------
;;; String Functions
;;;----------------------------------------------------------------------------


(defun str-exclude (x)
  "return nil if the argument is a character which should not appear in the
final string."
  (or (eq #\SPACE x)
      (eq #\" x)
      (eq #\, x)))


(defun str-tokenize (string)
  "return a list of tokens, ie #\SPACE-separated elements, from given string.
 Notice that encountered \"strings\" are included without double-quotes."
  (let ((nospc (lambda (x) (not (str-exclude x)))))
    (tokens string nospc 0)))


(defun str-first-token (string)
  "return the first encountered token in the given string"
  (string-right-trim
   '(#\SPACE)
   (block run
     (do* ((i 0 (1+ i))
           (c nil)
           (start nil)
           (len (length string))
```



```lisp
            (result (make-array len :element-type 'character
                                    :initial-element #\ ))
            (result-pos 0))
          ((>= i len) result)
        (setf c (char string i))
        (cond ((str-exclude c) (if start (return-from run result)))
              (t (setf start t)
                 (setf (char result result-pos) c)
                 (incf result-pos)))))))


(defun str-fetch-arg (com string)
  "return from the given string the token just after the given one"
  (let ((com-len (length com))
        (com-pos (search com string)))
    (cond (com-pos (str-first-token (subseq string (+ com-len com-pos))))
          (t nil))))


(defmacro str-append-to (str app)
  "appends the app string to the given string (side-effect)"
  `(setf ,str (concatenate 'string ,str ,app)))


(defun str-append (str app)
  "returns the concatenation of given strings"
  (concatenate 'string str app))


(defun str-coerce-sf (str)
  "returns a single-float from given string if possible"
  (coerce (read-from-string str)
          'single-float))


(defun str-coerce-r (str)
  "returns a real from given string if possible"
  (coerce (read-from-string str)
          'real))


;;;----------------------------------------------------------------------------
;;; Stream seeking Functions
;;;----------------------------------------------------------------------------


(defun stream-seek (pattern stream current-line)
  "Seeks in stream a given string pattern, if not present in the current-line string.
Returns the line of the stream where the pattern first occurs, NIL if end-of-line reached.
Side-effet on the given stream."
  (do ((line current-line (read-line stream nil 'eof)))
      ((eq line 'eof) nil)
    (if (search pattern line) (return line))))


(defun stream-multiline-seek (start-pattern stop-pattern-list stream current-line)
  "Extension to seek-in-stream: seeks between a start pattern and any stop-pattern,
commencing from the given current-line. Returns the new current-line as a first value,
the sought string as the second value."
  (do* ((line (stream-seek start-pattern stream current-line)
              (read-line stream nil 'eof))
        (acc nil)
```

```
        (str-seek (lambda (pat) (search pat line))))
      ((eq line 'eof) (values nil acc))
    (if (some #'identity (mapcar str-seek stop-pattern-list))
        (return (values line acc)))
    (str-append-to acc line)))
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: Object.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:13:58 $
;;; $Revision: 1.1.1.1 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 0)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 3)))


;;;---------------------------------------------------------------------------
;;; Type Declarations
;;;---------------------------------------------------------------------------


(deftype vertex-array ()
  `(simple-array vec4 (*)))

(deftype face-array ()
  `(simple-array glos-face (*)))


;;;---------------------------------------------------------------------------
;;; Vertex Functions
;;;---------------------------------------------------------------------------


(defun vertex-array-make (length)
  "build an array of null vertices"
  (make-array length
              :element-type 'vec4
              :initial-element nil))


;;;---------------------------------------------------------------------------
;;; Object type
;;;---------------------------------------------------------------------------


(defstruct (glos-object
             (:constructor obj-make)
             (:conc-name obj-)
             (:print-function obj-display))
  (name "Unnamed object" :type simple-base-string)
  (vertices #() :type vertex-array)
  (vertex-number 0 :type integer)
  (faces #() :type face-array)
  (fnormals #() :type (simple-array vec3 (*)))
  (vnormals #() :type (simple-array vec3 (*)))
  (face-number 0 :type integer))


(defun obj-display (o stream depth)
  (declare (ignore depth))
  (format stream "~%GLOS-Object: ~A ~%~A Vertices: ~%~A ~%~A Faces: ~%~A "
          (obj-name o)
          (obj-vertex-number o)
          (obj-vertices o)
          (obj-face-number o)
```



```lisp
          (obj-faces o)))


(defun obj-array-make (length)
  "build an array of null objects"
  (make-array length
              :element-type 'glos-object
              :initial-element nil))


;; calcule la normale a une face ...
(defun face-normal (a b c)
  "return the vec4 normal of the face defined by a b c"
  (let ((t1 (vec3-make (- #{a 0} #{b 0})
                       (- #{a 1} #{b 1})
                       (- #{a 2} #{b 2}))))
        (t2 (vec3-make (- #{c 0} #{b 0})
                       (- #{c 1} #{b 1})
                       (- #{c 2} #{b 2}))))
        (res (vec3-make 0.0 0.0 0.0)))
    (declare (type vec3 t1 t2 res))
    (vec-set-cross-product res t1 t2)
    (vec-set-normalize res)
    res))

(defun compute-normals (obj)
  "compute object normals"
  (setf (obj-fnormals obj) (make-array (obj-face-number obj) :element-type 'vec3
 :initial-element (vec3-make-null)))
  (setf (obj-vnormals obj) (make-array (obj-vertex-number obj) :element-type 've
c3 :initial-element (vec3-make-null)))
  ;; calcul des normales a chaque face
  (dotimes (i (obj-face-number obj))
    (multiple-value-bind (a b c) (face-to-vertices obj i)
      (setf (aref (obj-fnormals obj) i) (face-normal c b a))))
  ;; calcul des normales a chaque vertex
  (dotimes (i (obj-face-number obj))
    (let* ((face (aref (obj-faces obj) i))
           (ia (fac-a face))
           (ib (fac-b face))
           (ic (fac-c face))
           (ismooth (fac-smoothing face))
           (veca-sum (vec3-make-null))
           (vecb-sum (vec3-make-null))
           (vecc-sum (vec3-make-null))
           (nb-veca-sum 0)
           (nb-vecb-sum 0)
           (nb-vecc-sum 0))
      (dotimes (j (obj-face-number obj))
        (let* ((face2 (aref (obj-faces obj) j))
               (ja (fac-a face2))
               (jb (fac-b face2))
               (jc (fac-c face2))
               (jsmooth (fac-smoothing face2)))

          (when (= ismooth jsmooth)
            (when (or (= ia ja)
                      (= ia jb)
                      (= ia jc))
              (incf nb-veca-sum)
              (incf #{veca-sum 0}
                    #{(aref (obj-fnormals obj) j) 0})
```

```lisp
            (incf #{veca-sum 1}
                  #{(aref (obj-fnormals obj) j) 1})
            (incf #{veca-sum 2}
                  #{(aref (obj-fnormals obj) j) 2}))

          (when (or (= ib ja)
                    (= ib jb)
                    (= ib jc))
            (incf nb-vecb-sum)
            (incf #{vecb-sum 0}
                  #{(aref (obj-fnormals obj) j) 0})
            (incf #{vecb-sum 1}
                  #{(aref (obj-fnormals obj) j) 1})
            (incf #{vecb-sum 2}
                  #{(aref (obj-fnormals obj) j) 2}))

          (when (or (= ic ja)
                    (= ic jb)
                    (= ic jc))
            (incf nb-vecc-sum)
            (incf #{vecc-sum 0}
                  #{(aref (obj-fnormals obj) j) 0})
            (incf #{vecc-sum 1}
                  #{(aref (obj-fnormals obj) j) 1})
            (incf #{vecc-sum 2}
                  #{(aref (obj-fnormals obj) j) 2}))))
      (setf #{veca-sum 0} (/ #{veca-sum 0} nb-veca-sum))
      (setf #{veca-sum 1} (/ #{veca-sum 1} nb-veca-sum))
      (setf #{veca-sum 2} (/ #{veca-sum 2} nb-veca-sum))

      (setf #{vecb-sum 0} (/ #{vecb-sum 0} nb-vecb-sum))
      (setf #{vecb-sum 1} (/ #{vecb-sum 1} nb-vecb-sum))
      (setf #{vecb-sum 2} (/ #{vecb-sum 2} nb-vecb-sum))

      (setf #{vecc-sum 0} (/ #{vecc-sum 0} nb-vecc-sum))
      (setf #{vecc-sum 1} (/ #{vecc-sum 1} nb-vecc-sum))
      (setf #{vecc-sum 2} (/ #{vecc-sum 2} nb-vecc-sum))

      (vec-set-normalize veca-sum)
      (vec-set-normalize vecb-sum)
      (vec-set-normalize vecc-sum)

      (setf (aref (obj-vnormals obj) ia) veca-sum)
      (setf (aref (obj-vnormals obj) ib) vecb-sum)
      (setf (aref (obj-vnormals obj) ic) vecc-sum))))


;(defun object-to-vertices (obj)
;  "return the list of vertices of the faces of given object"
;  (let ((vxlist '())
;       (faces (obj-face-number obj)))
;    (compute-normals obj)
;    (dotimes (i faces (nreverse vxlist))
;      (multiple-value-bind (a b c) (face-to-vertices obj i)
;       (let* ((face (aref (obj-faces obj) i))
;              (ia (fac-a face))
;              (ib (fac-b face))
;              (ic (fac-c face))
;              ;(vf (aref (obj-fnormals obj) i))
;              (va (aref (obj-vnormals obj) ia))
;              (vb (aref (obj-vnormals obj) ib))
;              (vc (aref (obj-vnormals obj) ic)))
```



```lisp
;          ;;(push `(Normal ,#{vf 0} ,#{vf 1} ,#{vf 2}) vxlist)
;          (push `(Normal ,#{va 0} ,#{va 1} ,#{va 2}) vxlist)
;          (push `(Vertex ,#{a 0} ,#{a 1} ,#{a 2} ,#{a 3}) vxlist)
;          (push `(Normal ,#{vb 0} ,#{vb 1} ,#{vb 2}) vxlist)
;          (push `(Vertex ,#{b 0} ,#{b 1} ,#{b 2} ,#{b 3}) vxlist)
;          (push `(Normal ,#{vc 0} ,#{vc 1} ,#{vc 2}) vxlist)
;          (push `(Vertex ,#{c 0} ,#{c 1} ,#{c 2} ,#{c 3}) vxlist))))))

;(defun object-to-vertices (obj)
;  "return the list of vertices of the faces of given object"
;  (let ((vxlist '())
;       (faces (obj-face-number obj)))
;    (dotimes (i faces (nreverse vxlist))
;      (multiple-value-bind (a b c) (face-to-vertices obj i)
;       (push a vxlist)
;       (push b vxlist)
;       (push c vxlist)))))


;;;---------------------------------------------------------------------------
;;; Face type
;;;---------------------------------------------------------------------------


(defstruct (glos-face
            (:constructor fac-make)
            (:conc-name fac-)
            (:print-function fac-display))
  (A 0 :type fixnum)
  (B 0 :type fixnum)
  (C 0 :type fixnum)
  (AB 0 :type fixnum)
  (BC 0 :type fixnum)
  (CA 0 :type fixnum)
  (smoothing 0 :type fixnum))


(defun fac-display (f stream depth)
  (declare (ignore depth))
  (format stream "~%GLOS-Face: ~%A:~A B:~A C:~A ~%AB:~A BC:~A CA:~A ~%Smoothing ~A"
          (fac-a f) (fac-b f) (fac-c f)
          (fac-ab f) (fac-bc f) (fac-ca f)
          (fac-smoothing f)))


(defun face-array-make (length)
  "build an array of null faces"
  (make-array length
              :element-type 'glos-face
              :initial-element nil))


(defun face-to-vertices (obj face-num)
  "translate the n-th face of the given object into vertices"
  (let* ((face (aref (obj-faces obj) face-num))
         (a (fac-a face))
         (b (fac-b face))
         (c (fac-c face))
         (vx1 (aref (obj-vertices obj) a))
         (vx2 (aref (obj-vertices obj) b))
         (vx3 (aref (obj-vertices obj) c)))
    (values vx1 vx2 vx3)))
```

```lisp
;;; ---------------------------------------------------------------------------
;;; $RCSfile: ASCParser.lisp,v $
;;; $Author: hubert $
;;; $Date: 2000/05/05 06:42:56 $
;;; $Revision: 1.2 $
;;; ---------------------------------------------------------------------------

(declaim (optimize (speed 0)
                   (compilation-speed 0)
                   (safety 0)
                   (debug 3)))


;;;----------------------------------------------------------------------------
;;; Parser Constants
;;;----------------------------------------------------------------------------


(defconstant +code-name+ "Named object:")
(defconstant +code-vx-num+ "Vertices:")
(defconstant +code-fa-num+ "Faces:")
(defconstant +code-vx-list+ "Vertex list:")
(defconstant +code-fa-list+ "Face list:")
(defconstant +code-smoothgroup+ "Smoothing:")
(defconstant +code-vx+ "Vertex ")
(defconstant +code-fa+ "Face ")
(defconstant +code-vx-coords+ '("X:" "Y:" "Z:"))
(defconstant +code-fa-kwds+ '("A:" "B:" "C:" "AB:" "BC:" "CA:"))


;;;----------------------------------------------------------------------------
;;; Parser functions
;;;----------------------------------------------------------------------------


(defun asc-parse-file (filename)
  "Run through the given ASC file and returns a list of objects"
  (let ((path (make-pathname :name filename)))
    (with-open-file
     (stream path :direction :input)
     (do* ((current-line "")            ;current line of the stream
           (object-list '()))
         ((equal current-line nil) (nreverse object-list))
       (multiple-value-bind (object line) (asc-read-object stream current-line)
         (setf current-line line)
         (push object object-list))))))


(defun asc-read-object (stream current-line)
  "Read the current open stream and its current line.
Returns the next object as 1st value, the new current line as second value.
Side-effect on the stream."
  (let* ((name nil)
         (vertex-num nil)
         (face-num nil)
         (smoothgrp 0)
         (vertex-array nil)
         (face-array nil)
         (result nil)                   ;result of last multiline-seek
         (single-seek (lambda (pat) (setf current-line
                                          (stream-seek pat stream current-line))
))
```



```lisp
         (multiline-seek
          (lambda (x) (let* ((pat (mkstr +code-fa+ x ":"))
                             (stop-pat-lst (list (mkstr +code-fa+ (+ x 1) ":") +
code-name+)))
                        (multiple-value-bind (c-line results)
                            (stream-multiline-seek pat stop-pat-lst stream curre
nt-line)
                          (setf current-line c-line) results))))
         (fetch-sf-cline (lambda (comname) (str-coerce-sf (str-fetch-arg comname
 current-line))))
         (fetch-r-result (lambda (comname) (let ((x (str-fetch-arg comname resul
t)))
                                             (when x (str-coerce-r x))))))


    (block read-object
      ;;recuperate the minimal object ID
      (ignore-errors
        (funcall single-seek +code-name+)
        (setf name (string-upcase (str-fetch-arg +code-name+ current-line)))
        (funcall single-seek +code-vx-num+)
        (setf vertex-num (str-coerce-r (str-fetch-arg +code-vx-num+ current-line
)))
        (setf face-num (str-coerce-r (str-fetch-arg +code-fa-num+ current-line))
))

      ;;exit now if the minimal information cannot be fetched
      (when (not (and name vertex-num face-num))
        (return-from read-object (values nil current-line)))

      ;;get an array of vertices
      (when (> vertex-num 0)
        (setf vertex-array (vertex-array-make vertex-num))
        (funcall single-seek +code-vx-list+)
        (dotimes (i vertex-num)
          (funcall single-seek (mkstr +code-vx+ i ":"))
          (setf (aref vertex-array i)
                (multiple-value-bind (x y z)
                    (list-values (mapcar fetch-sf-cline +code-vx-coords+))
                  (vec4-make x y z 1.0)))))

      ;;get an array of faces
      (when (> face-num 0)
        (setf face-array (face-array-make face-num))
        (funcall single-seek +code-fa-list+)
        (dotimes (i face-num)
          (setf result (funcall multiline-seek i))
          (setf smoothgrp (funcall fetch-r-result +code-smoothgroup+))
          (unless smoothgrp (setf smoothgrp 0))
          (setf (aref face-array i)
                (multiple-value-bind (a b c ab bc ca)
                    (list-values (mapcar fetch-r-result +code-fa-kwds+))
                  (fac-make :A a :B b :C c :AB ab :BC bc :CA ca :smoothing smoot
hgrp)))))

      ;;return the fetched object and the current stream-line
      (values (obj-make :name name
                        :vertices vertex-array
                        :vertex-number vertex-num
                        :faces face-array
                        :face-number face-num)
              current-line))))
```

```
;;;-----------------------------------------------------------------------------
;;; Functions generating "Begin" expressions from objects
;;;-----------------------------------------------------------------------------

(defun Begin-object (obj)
  (declare (optimize (speed 3)
                     (compilation-speed 0)
                     (safety 0)
                     (debug 0)))
  (Begin :TRIANGLES
         (let ((nbface (obj-face-number obj)))
           (declare (type fixnum nbface))
           (do ((i 0 (1+ i)))
               ((>= i nbface))
             (declare (type fixnum i))
             (let* ((face (aref (obj-faces obj) i))
                    (ia (fac-a face))
                    (ib (fac-b face))
                    (ic (fac-c face))
                    ;;(vf (aref (obj-fnormals obj) i))
                    (pa (aref (obj-vertices obj) ia))
                    (pb (aref (obj-vertices obj) ib))
                    (pc (aref (obj-vertices obj) ic))
                    (va (aref (obj-vnormals obj) ia))
                    (vb (aref (obj-vnormals obj) ib))
                    (vc (aref (obj-vnormals obj) ic)))
               (declare (type vec4 pa pb pc)
                        (type vec3 va vb vc))
               ;;(Normal #{vf 0} #{vf 1} #{vf 2})
               (Normal #{va 0} #{va 1} #{va 2})
               (Vertex #{pa 0} #{pa 1} #{pa 2} #{pa 3})
               (Normal #{vb 0} #{vb 1} #{vb 2})
               (Vertex #{pb 0} #{pb 1} #{pb 2} #{pb 3})
               (Normal #{vc 0} #{vc 1} #{vc 2})
               (Vertex #{pc 0} #{pc 1} #{pc 2} #{pc 3}))))))

(defun load-list-asc (filename)
  (let ((obj-list (asc-parse-file filename)))
    (mapcar #'compute-normals obj-list)
    obj-list))
```