Chapter 9.
Scheme Data Types

Topic: Ignore

ACIS provides Scheme data types (Scheme objects) specifically for use with ACIS Scheme extensions, in addition to those that are native to the Scheme language. Some of the native Scheme data types are documented if they are needed to describe the ACIS data types. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

acis–journal

Scheme Data Type: ACIS Journal

Description: An acis–journal data type is used to control the journaling operation.

Derivation: acis–journal : scheme–object

C++ Type: AcisJournal

External Rep: #[file: "%s" { enabled | disabled }]

Example:

```
; acis–journal (data type)
; set a journal file
(define j
    (acis_journal:set "file"
    "sweep_journal_example"))
;; j
; set version info
(define v (versiontag 7 0 0))
;; v
; Create acis–options using journal and version info
(define ao
    (acisoptions:set "journal" j "version" v))
;; ao
(define b1
    (solid:block (position 0 0 0) (position 3 1 1)))
;; b1
```

```
(define b2
    (solid:block (position 2 0 0) (position 3 6 1)))
;; b2
(define b3
    (solid:block (position 0 5 0) (position 3 6 1)))
;; b3
(define b4
    (solid:block (position 0 0 0) (position 1 6 1)))
;; b4
(zoom–all)
;; #[view 1049866]
; Start the journaling operation
(acis_journal:start ao)
;; #t
(define u (bool:unite b1 b2 ao))
;; u
; Pause the journaling operation
(acis_journal:pause ao)
;; #t
(define u1 (bool:unite b1 b3 ao))
;; u1
; Resume the journaling operation
(acis_journal:resume ao)
;; #t
(define u2 (bool:unite b1 b4 ao))
;; u2
; End the journaling operation
(acis_journal:end ao)
;; #t
```

acis–options

Scheme Data Type: ACIS Journal, History and Roll

Description: An acis–options data type is used to set options related to journaling and versioning.

Derivation: acis–options : scheme–object

C++ Type: AcisOptions

External Rep: #[journal–"%s" { enabled | not–enabled } | version – %d]

Example:

```
; acis-options (data type)
; Clear the part
(part:clear)
; Set journal file name
(define j
    (acis_journal:set "file"
    "sweep_journal_example"))
; Create a version tag
(define v (versiontag 7 0 0))
; Create acis-options with journal and version info
(define ao (acisoptions:set "journal" j "version" v))
(define profile
    (solid:block (position 0 0 0) (position 2 2 0)))
(define path
    (wire-body:points (list (position 0 0 0)
    (position 0 0 2) (position 1 1 4))))
(define opts (sweep:options))
; Start the journaling operation using acis-options
(acis_journal:start ao)
(sweep:law profile path opts ao)
; End the journaling operation
(acis_journal:end ao)
```

adm–options

Scheme Data Type: History and Roll

Description: An adm–options data type overrides acis–options algorithmic versioning for specific behaviors: the use of boundary loads, and automatic surface trimming.

Derivation: adm–options : scheme–object

C++ Type: adm_options

External Rep: #[Adm_Options]

Example:

```
; adm–options (data type)
; Clear the part
(part:clear)
(solid:block 0 0 0 10 10 10)
(ray:queue 36.5075 –317.674 384.56
    –0.0619225 0.65135 –0.756246 1)
(define my–face (pick–face))
(define admo (ds:adm–options ”use_boundary_loads” 0))
; start adm using boundary constraints
; instead of boundary loads
(ds:start–adm my–face admo)
```

animation–figure

Scheme Data Type: Animation

Description: An animation–figure is a type of rubberband driver used to apply transformations to model objects over time to create motion.

Derivation: animation–figure : rbd–driver : scheme–object

C++ Type: animation_figure

External Rep: #[rbd–driver %x]
where the hex number is the driver address.

Example:

```
; animation–figure (data type)
; Create an animation figure.
(define my_block (solid:block (position –20 –20 –20)
    (position 20 20 20)))
;; my_block
(define my_fig (afig:create my_block))
;; my_fig
(afig:show my_fig)
;; ()
(define my_trans (transform:rotation (position 0 0 0)
    (gvector 0 1 0) 1))
;; my_trans
(let loop ((i 0))
    (if (< i 360) (begin
        (afig:apply–transform my_fig my_trans)
        (loop (+ i 1))
    )))
;; ()
```

attribute

Scheme Data Type: Attributes

Description: An attribute is a general–purpose data entity that attaches to other entities to record user or other information. attribute objects are saved and restored as part of the model.

Derivation: attribute : entity : scheme–object

C++ Type: NAMED_ATTRIB

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; attribute (data type)
; Create, add, modify, inquire, and
; delete an attribute.
(define my_block (solid:block (position 0 0 0)
    (position 10 23 45)))
;; my_block
(attrib:add my_block ”density” 12.574)
;; ()
(attrib:get my_block ”density”)
;; ((”density” . 12.574))
(attrib:replace my_block ”density” 12.555)
;; ()
(attrib:get my_block ”density”)
;; ((”density” . 12.555))
(attrib:remove my_block ”density”)
;; ()
```

background

Scheme Data Type: Backgrounds and Foregrounds

Description: A background is an entity that specifies the background shader and its arguments to be used during rendering. A background shader places a color pattern over all pixels not obscured by the model. Background types are “plain”, “graduated”, or “clouds”. background objects are saved and restored as part of the model.

Derivation: background : entity : scheme–object

C++ Type: RH_BACKGROUND

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; background (data type)
; Define and inquire a background object.
(define my_bg1 (background ”clouds”))
;; my_bg1
(background? my_bg1)
;; #t
```

bezier–edge

Scheme Data Type: Model Geometry, Model Object

Description: A bezier–edge is a topological entity that describes a cubic Bezier curve using four control points. bezier–edge objects are saved and restored as part of the model.

Derivation: bezier–edge : edge : entity : scheme–object

C++ Type: EDGE–>CURVE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; bezier-edge (data type)
; Create bezier-edge 1.
(define my_edge (edge:bezier (position 10 0 0)
    (position 10 0 30) (position 30 0 30)
    (position 30 0 0)))
;; my_edge
```

body

Scheme Data Type: Model Topology, Model Object

Description: A body is the highest level topological entity, and it can be a wire body, a solid body, or mixed body. Wire bodies contain wires, coedges, edges, and vertices. Solid bodies contain lumps, shells, subshells, faces, loops, coedges, edges, and vertices. Mixed (solid and wire) bodies contain lumps, shells, subshells, faces, loops, coedges, edges, vertices, and wires. body objects are saved and restored as part of the model.

Derivation: body : entity : scheme–object

C++ Type: BODY

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; body (data type)
; Define and inquire a solid body object.
(define my_block1 (solid:block (position 0 0 0)
    (position 5 10 15)))
;; my_block1
(body? my_block1)
;; #t
```

boolean

Scheme Data Type: Booleans

Description: A boolean is a native Scheme data type having either the value #t (true) or the value #f (false). It represents a logical, or Boolean, value.

Derivation: boolean : scheme–object

C++ Type: logical

External Rep: #t or #f

Example:

```
; boolean (data type)
; Define and inquire a boolean object.
(define ON #t)
;; ON
(define OFF #f)
;; OFF
(boolean? ON)
;; #t
ON
;; #t
OFF
;; #f
```

cell

Scheme Data Type: Cellular Topology

Description: A cell attaches cellular topology data to each lump within each body. cell objects are saved and restored as part of the model.

Derivation: cell : entity : scheme–object

C++ Type: CELL

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; cell? (data type)
; Create a solid block, attach cellular topology to
; the lumps and determine if the lump is a cell.
(define my_block (solid:block (position –20 –20 –20)
    (position 20 20 20)))
;; my_block
(define my_cell (cell:attach my_block))
;; my_cell
(cell? (car my_cell))
;; #t
```

circular–curve

Scheme Data Type: Construction Geometry

Description: A circular–curve is a data structure containing a curve that is circular in nature. It is not derived from entity, and thus is a lighter weight object. It is often used for evaluation purposes when a curve needs to be created but not saved as part of the geometric model.

Derivation: circular–curve : curve : scheme–object

C++ Type: bounded_arc

External Rep: #[curve %x]
where the hexadecimal number is the memory location of the curve. This visually identifies a particular curve, but it cannot be used in Scheme to directly access the curve. Instead, curves should be defined, then the defined name accesses the curve.

Example:

```
; circular–curve (data type)
; Define and inquire a circular–curve object.
(define my_curve1 (curve:circular (position 0 0 0) 25
    (gvector 0 1 0)))
;; my_curve1
(curve:circular? my_curve1)
;; #t
```

circular–edge

Scheme Data Type: Model Object, Model Geometry

Description: A circular–edge is an entity containing a circular–curve. circular–edge objects are saved and restored as part of the model.

Derivation: circular–edge : curve–edge : edge: entity : scheme–object

C++ Type: EDGE–>ELLIPSE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; circular–edge (data type)
; Define and inquire a circular–edge object.
(define edge1 (edge:circular (position 0 0 0)
    30 0 90))
;; edge1
(edge:circular? edge1)
;; #t
```

coedge

Scheme Data Type: Model Topology, Model Object

Description: A coedge is a topological entity that records the occurrence of an edge in a loop of a face. Coedges permit edges to occur in one, two, or more faces, and so makes possible the modeling of sheets and solids (manifold or not). A loop refers to one coedge in the loop, from which pointers lead to the other coedges of the loop. Coedges are generated automatically when a geometric object is created. There is no way to explicitly create a coedge. coedge objects are saved and restored as part of the model.

Derivation: coedge : entity : scheme–object

C++ Type: COEDGE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
;; coedge (data type)
;; Create a block an inquire its coedges.
(define my_block (solid:block (position 0 0 0)
    (position 10 10 10)))
;; my_block
(define my_coedges (entity:coedges my_block))
;; my_coedges
; (#[entity 2 1] #[entity 3 1] #[entity 4 1]
; #[entity 5 1] #[entity 6 1] #[entity 7 1]
; #[entity 8 1] #[entity 9 1] #[entity 10 1]
; #[entity 11 1] #[entity 12 1] #[entity 13 1]
; #[entity 14 1] #[entity 15 1] #[entity 16 1]
; #[entity 17 1] #[entity 18 1] #[entity 19 1]
; #[entity 20 1] #[entity 21 1] ...)
(coedge? (car my_coedges))
;; #t
```

color

Scheme Data Type: Colors

Description: A color data type specifies the display color for rendering and other operations. Colors are specified with three real numbers corresponding to red, green, and blue. The numbers should be normalized (between 0 and 1). Extensions requiring a color argument accept a color object or a single integer ranging from 0 to 7, corresponding to one of the eight base colors normally used. Typically color objects are created using the color:rgb extension.

Derivation: color : scheme–object

C++ Type: rgb_color

External Rep: #[color %g %g %g]
where the first double is the red component,
the second double is the green component,
and the third double is the blue component.

Example:

```
; color (data type)
; Define and inquire a red color object
(define red (color:rgb 1 0 0))
;; red
(color:rgb? red)
;; #t
```

conic–edge

Scheme Data Type: Model Object, Model Geometry

Description: A conic–edge is a topological entity that describes a rho conic edge in which the geometrical definition represents a hyperbola or a parabola. conic–edge objects are saved and restored as part of the model.

Derivation: conic–edge : edge : entity : scheme–object

C++ Type: EDGE–>CURVE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; conic–edge (data type)
; Create rho conic parabola.
(define my_edge (edge:conic (position –50 0 0)
    (position 50 0 0) (position 0 50 0) 0.5))
;; my_edge
```

conical–face

Scheme Data Type: Model Object, Model Geometry

Description: A conical–face is a geometric entity that is a face that is conical in nature. conical–face objects are saved and restored as part of the model.

Derivation: conical–face : face : entity : scheme–object

C++ Type: FACE–>CONE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; conical–face (data type)
; Define and inquire a conical–face object.
(define my_cyl (solid:cylinder (position 0 0 0)
    (position 25 25 0) 30))
;; my_cyl
(define my_faces (entity:faces my_cyl))
;; my_face
; (#[entity 2 1] #[entity 3 1] #[entity 4 1])
(conical–face (car my_faces))
;; #f
(face? (car my_faces))
;; #t
```

curve

Scheme Data Type: Construction Geometry, Model Geometry

Description: A curve is a data structure used for evaluation purposes and for storing curve data within an entity. The curve is a base from which line, arc, ellipse, and spline are derived. curve objects are not saved and restored as part of the model.

Consider each curve as a parametric curve that maps an interval of the real line into a 3D vector space (object space). The mapping is continuous and one–to–one, except for closed curves. The curve is assumed to have a continuous first derivative whose length is bounded above and below by nonzero constants.

Derivation: curve : scheme–object

C++ Type: bounded_curve

External Rep: #[curve %x]
where the hexadecimal number is the memory location of the curve. This visually identifies a particular curve, but it cannot be used in Scheme to directly access the curve. Instead, curves should be defined, then the defined name accesses the curve.

Example:

```
; curve (data type)
; Create and inquire a curve object.
(define curve1 (curve:circular (position 0 0 0) 25
    (gvector 0 1 0)))
;; curve1
(curve? curve1)
;; #t
```

curve–edge

Scheme Data Type: Model Geometry, Model Object

Description: A curve–edge is a topological entity that is an edge represented by a curve. curve–edge objects are saved and restored as part of the model.

Derivation: curve–edge : edge : entity : scheme–object

C++ Type: EDGE–>CURVE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; curve–edge (data type)
; Create and inquire a curve–edge object.
(define my_edge (curve–>edge
    (curve:circular (position 0 0 0) 25))))
;; my_edge
(edge:curve? my_edge)
;; #t
(edge? my_edge)
;; #t
```

cylindrical–face

Scheme Data Type: Model Geometry, Model Object

Description: A cylindrical–face is a geometric entity that is a face that is cylindrical in nature. cylindrical–face objects are saved and restored as part of the model.

Derivation: cylindrical–face : face : entity : scheme–object

C++ Type: FACE–>CONE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; cylindrical–face (data type)
; Create and inquire a cylindrical–face object.
(define my_cyl (solid:cylinder (position 0 0 0)
    (position 25 25 0) 30))
;; my_cyl
(define my_faces (entity:faces my_cyl))
;; my_faces
(face:cylindrical? (car my_faces))
;; #t
```

dl–item

Scheme Data Type: Viewing

Description: A dl–item is a scheme–object that records a particular item, such as a point, polyline, or text, within the display list. dl–item objects are not saved and restored as part of the model.

Derivation: dl–item : scheme–object

C++ Type: DL_item

External Rep: #[dl–item %p]
where the pointer points to the curve.

Example:
```
; dl–item (data type)
; Create and inquire a display list item.
(define dp (dl–item:point (position 0 0 0)))
;; dp
(dl–item? dp)
;; #t
```

edge

Scheme Data Type: Model Topology, Model Object

Description: An edge is a topological entity associated with a curve. An edge is bounded by one or more vertices, and refers to one vertex at each end. If the reference at either or both ends is NULL, the edge is unbounded in that direction. Each edge contains a record of its sense (FORWARD or REVERSED) relative to its underlying curve. edge objects are saved and restored as part of the model.

Derivation: edge : entity : scheme–object

C++ Type: EDGE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; edge (data type)
; Create and inquire an edge.
(define my_edge (edge:circular (position 0 0 0)
    25 0 185))
;; my_edge
(edge? my_edge)
;; #t
```

elliptical–curve

Scheme Data Type: Construction Geometry

Description: An elliptical–curve is a data structure containing a curve that is elliptical in nature. It is not derived from entity, and thus is a lighter weight object. It is often used for evaluation purposes when a curve needs to be created but not saved as part of the geometric model.

Derivation: elliptical–curve : curve : scheme–object

C++ Type: bounded_arc

External Rep: #[curve %x]
where the hexadecimal number is the memory location of the curve. This visually identifies a particular curve, but it cannot be used in Scheme to directly access the curve. Instead, curves should be defined, then the defined name accesses the curve.

Example:
```
; elliptical–curve (data type)
; Define an elliptical–curve object.
(define my_edge (edge:elliptical (position 15 15 0)
    (position 25 15 0) 2 0 270))
;; my_edge
(curve:from–edge my_edge)
;; #[curve 401ac8b8]
```

elliptical–edge

Scheme Data Type: Model Geometry, Model Object

Description: An elliptical–edge is an edge that is elliptical in nature. elliptical–edge objects are saved and restored as part of the model.

Derivation: elliptical–edge : edge : entity : scheme–object

C++ Type: EDGE–>ELLIPSE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; elliptical–edge (data type)
; Create and inquire an elliptical–edge object.
(define my_edge (edge:elliptical (position 15 15 0)
    (position 25 15 0) 2 0 270))
;; my_edge
(edge:elliptical? my_edge)
;; #t
```

entity

Scheme Data Type: Entity, Model Object

Description: An entity is a top–level object from which all other objects representing permanent objects in ACIS, such as geometric, topological, attribute, and transform objects, are derived. It does not represent any specific object within the modeler. Instead, it represents common data and functionality that must be contained in all classes that represent permanent objects within the modeler. entity objects are saved and restored as part of the model.

An entity may also contain pointers from objects to system–defined and user–defined attributes. Not all objects use attribute pointers, but automatic creation and deletion of attributes for any object is supported.

A deleted entity is an entity that has been deleted but is still referenced by some other scheme–object or procedure. Deleted entities cannot be directly used. However, if the state of the model is rolled back past the point of deletion, the entity returns and can be used again.

Derivation: entity : scheme–object

C++ Type: ENTITY

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

#[(deleted) entity %d]
where the integer is the entity ID.

Example:

```
; entity (data type)
; Create and inquire an entity object.
(define my_torus (solid:torus
    (position –10 –10 –10) 7 3))
;; my_torus
(entity? my_torus)
;; #t
```

entity–filter

Scheme Data Type: Filtering, Picking

Description: An entity–filter is a procedural object that selects entities from an entity–list. Complex filtering is generated by combining basic color, type, and other filters using “and”, “or”, and “not” entity filters.

Derivation: entity–filter : scheme–object

C++ Type: entity_filter

External Rep: #[entity–filter %x]
where the hexadecimal number is the memory location of the filter. This memory location should not directly reference the filter. Instead, the filter should be named using define, then referenced using the name.

Example:
```
; entity–filter (data type)
; Create and inquire an entity–filter object that
; finds only entities that are displayed.
(env:set–auto–display #f)
;; ()
(define my_block (solid:block (position 0 0 0)
    (position 20 30 40)))
;; my_block
(define my_edge (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; my_edge
(define my_edge2 (edge:circular (position 0 0 0) 20))
;; my_edge2
(env:set–auto–display #t)
;; ()
(define my_sph (solid:sphere (position 20 30 40) 30))
;; my_sph
(define my_cyl (solid:cylinder (position 40 40 0)
    (position 40 40 40) 10))
;; my_cyl
(filter:apply (filter:display) (part:entities))
;; (#[entity 5 1] #[entity 6 1])
```

entray

Scheme Data Type: Picking

Description: An entray is a composition of an entity and a ray. The first element is an entity. The second element is a ray that consists of a position and a gvector. entray objects are not saved and restored as part of the model.

Derivation: entray : entity, ray : scheme–object

C++ Type: ENTITY, pick_ray

External Rep: #[entity–with–ray %d %d (%g %g %g) (%g %g %g)]
where the first integer is the entity ID,
the second integer is the part ID,
the first triplet of double values is the ray's (x y z) position,
and the second triplet of double values is the ray's (x y z) gvector.

Example:
```
; entray (data type)
; Create and inquire an entray object.
(define my_block (solid:block (position –35 –35 –35)
    (position 15 15 15)))
;; my_block
(define my_entray (entray my_block
    (ray (position 0 0 0) (gvector 0 0 1))))
;; my_entray
(entray? my_entray)
;; #t
```

environment–map

Scheme Data Type: Environment Maps

Description: An environment–map is an entity used for rendering of reflections. It consists conceptually of a six–sided cube of image data that is wrapped around reflective objects during rendering. It is created from image data files, from rendering the model, or from a procedure that generates image data. Environment–mapped reflections are not visible in the Basic Rendering Component, but are supported for compatibility with the Advanced Rendering Component. environment–map objects are saved and restored as part of the model.

Derivation: environment–map : entity : scheme–object

C++ Type: RH_ENVIRONMENT_MAP

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; environment–map (data type)
; Create and inquire an environment–map object.
(define map1 (environment–map:stripe 5 20 30))
;; map1
(environment–map? map1)
;; #t
```

face

Scheme Data Type: Model Topology, Model Object

Description: A face is a topological entity that is a portion of a single geometric surface. One or more loops of edges bound a face. Faces are open or closed. A face with no loops occupies the entire surface, finite or infinite, on which the face lies. Thus a face may stand for an infinite plane or for a complete sphere. Each face records its sense relative to its underlying surface (same sense or opposite sense). face objects are saved and restored as part of the model.

Derivation: face : entity : scheme–object

C++ Type: FACE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; face (data type)
; Create and inquire a face object.
(define my_block (solid:block (position 0 0 0)
    (position –25 –25 –25)))
;; my_block
(define my_faces (entity:faces my_block))
;; my_faces
; (#[entity 2 1] #[entity 3 1] #[entity 4 1]
; #[entity 5 1] #[entity 6 1] #[entity 7 1])
(face? (car my_faces))
;; #t
```

foreground

Scheme Data Type: Backgrounds and Foregrounds

Description: A foreground is an entity that specifies the foreground shader and its arguments to be used during rendering. A foreground shader places a color pattern over all pixels. Foreground types are “none”, “depth cue”, or “fog”. foreground objects are saved and restored as part of the model.

Derivation: foreground : entity : scheme–object

C++ Type: RH_FOREGROUND

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; foreground (data type)
; Define and inquire a foreground object.
(define fg1 (foreground ”fog”))
;; fg1
(foreground? fg1)
;; #t
```

glue–options

Scheme Data Type: Booleans

Description: A glue–options object is to be used in conjunction with two bodies (blank and tool) whose intersection is known to lie along a set of coincident faces. See documentation for api_boolean_glue for the definition of coincident faces.

Derivation: glue–options : scheme–object

C++ Type: logical

External Rep: #t or #f

Example:
```
; glue–options (data type)
; Define and inquire a boolean object.
```

graph

Scheme Data Type: Graph Theory

Description: A graph is used to implement graph theory in ACIS. A graph is composed of vertices and edges. Edges are represented as a dash (–) between a pair of vertices. Vertices are represented as strings. Cells or faces of a model can be mapped into a graph structure. Each cell or model face is represented as a vertex, while connectivity between cells or model faces are represented as edges. There are several graph operations available for ordering and querying the graph structure.

Derivation: graph : scheme–object

C++ Type: generic_graph

External Rep: #[graph “...”]
where strings represent vertices of the graph and dashes (–) between pairs of vertices represent edges of the graph.

Example:

```
; graph (data type)
; Create a simple example
(define g1 (graph "me–you us–them"))
;; g1
; g1
; Create an example using entities.
(define b1 (solid:block (position –5 –10 –20)
    (position 5 10 15)))
;; b1
; b1
(define faces1 (entity:faces b1))
;; faces1
; faces1 => (#[entity 3 1] #[entity 4 1]
; #[entity 5 1] #[entity 6 1] #[entity 7 1]
; #[entity 8 1])
; Turn the block faces into vertices of the graph.
(define g3 (graph faces1))
;; g3
; g3 => #[graph "(Face 5)–(Face 4) (Face 4)–(Face 3)
; (Face 5)–(Face 2) (Face 3)–(Face 2)
; (Face 4)–(Face 1) (Face 3)–(Face 1)
; (Face 2)–(Face 1) (Face 5)–(Face 1)
; (Face 2)–(Face 0) (Face 3)–(Face 0)
; (Face 4)–(Face 0) (Face 5)–(Face 0)"]
```

gvector

Scheme Data Type: Mathematics, Scheme Interface

Description: A gvector is a composition of three real numbers representing the *x*, *y*, and *z* components of a 3D vector. This is an ACIS–defined Scheme data type used to represent a (mathematical) vector with magnitude and direction. It is named gvector only to differentiate it from the inherent Scheme data type vector, which represents an array.

Derivation: gvector : scheme–object

C++ Type: SPAvector

External Rep: #[gvector %.15g %.15g %.15g]
where the first double number is the gvector's *x* component,
the second double number is the gvector's *y* component,
and the third double number is the gvector's *z* component.

Example:

```
; gvector (data type)
; Define and inquire a gvector object.
(define x–vector (gvector 1 0 0))
;; x–vector
(gvector? x–vector)
;; #t
```

history

Scheme Data Type: History and Roll

Description: Two types of history exist. Part histories contain rollback information for all entities in a part, unless those entities have entity history. Finally, the default history contains history for entities without history and in a part without part history.

Derivation: history: scheme–object

C++ Type: HISTORY_STREAM

External Rep: #[history %d]
where the number is the history ID number.

Example:

```
; history (data type)
; Create a new part and get its history.
(define my_part1 (part:new))
;; my_part1
(history my_part1)
;; #[history 0 1]
```

phlv5–data

Scheme Data Type: Hidden Line Removal

Description: Hidden line data.

Derivation: phlv5–data:scheme–object

C++ Type: phlv5_data

External Rep: #[phlv5–data %x]

Example:

```
; phlv5–data
; create a body
(define block (solid:block (position 0 0 0)
    (position 10 10 10)))
;; block
(iso)
;; #[view 395052]
(zoom–all)
;; #[view 395052]
(define data (phlv5:compute block 1))
;; data
(part:save ”phlv5.sat”)
;; #t
(part:clear)
;; #t
(define entities (part:load ”phlv5.sat”))
;; entities
(define data (phlv5:retrieve (car (part:entities))
1))
;; data
(phlv5:draw data)
;; #[phlv5–data 8839cd8]
(phlv5:clean 1 entities)
;; ()
(define data (phlv5:retrieve (car (part:entities))
1))
;; data
(phlv5:draw data)
;; #[phlv5–data 8839d58]
```

phlv5–options

Scheme Data Type: Hidden Line Removal

Description: Used to specify options for hidden line removal.

Derivation: phlv5–options: scheme–object

C++ Type: phlv5_options

External Rep: #[phlv5–options %x, “hidden_line_style:” %s, “sag_resolution:” %d “resolution:” %d]

Example:

```
; phlv5–options
; create a body
(define block (solid:block (position 0 0 0)
    (position 10 10 10)))
;; block
(iso)
;; #[view 657196]
(zoom–all)
;; #[view 657196]

(define opts
    (phlv5:options ”hidden_line_style” ”dashed”))
;; opts
(phlv5:compute opts)
;; #f
```

ihl–data

Scheme Data Type: Interactive Hidden Line

Description: A ihl–data object contains interactive hidden line data generated by the ihl:compute or ihl:retrieve extensions.

Derivation: ihl–data : scheme–object

C++ Type: ihl_data

External Rep: #[ihl–data %x]

Example:

```
; ihl–data (datatype)
; Create interactive hidden line data.
(define my_block (solid:block (position –10 –25 –35)
    (position 10 25 35)))
;; my_block
; my_block => #[entity 2 1]
(define my_cyl (solid:cylinder (position 0 0 –20)
    (position 0 0 20) 30))
;; my_cyl
; my_cyl => #[entity 3 1]
(define my_combo (bool:unite my_block my_cyl))
;; my_combo
; my_combo => #[entity 2 1]
(define ihldata1 (ihl:compute 1 my_combo #f))
;; ihldata1
; ihldata1 => #[ihl–data 40248e60]
```

integer

Scheme Data Type: Mathematics

Description: An integer object is a Scheme language primitive containing a single integer value. An integer is always a real. Not all reals are integers. integer objects are saved and restored as part of the model only when they are part of an entity.

Derivation: integer : scheme–object

C++ Type: int

External Rep: %d

Example:

```
; integer (data type)
; Define and inquire an integer object.
(define five 5)
;; five
(integer? five)
;; #t
```

law

Scheme Data Type: Laws

Description: A law is a Scheme data type that holds a pointer to a law C++ class. The low–level implementation of laws in Scheme uses API's and law string parsing to create C++ classes from law function strings enclosed in quotation marks.

The law functions are very similar to common mathematical notation and to the adaptation of mathematical notation for use in computers. The valid syntax for the character strings are given in the law function templates.

The strings used to define laws are not case–sensitive, but when returned from a law function, the lower–case letters are converted to upper–case letters. Only laws that are used to define the geometry, such as wire–body:offset and sweep:law, are stored in a save file

Derivation: integer : scheme–object

C++ Type: law

External Rep: #[law "%s"]
where the quoted string represents a valid law made up of law functions.

Example:

```
; law (data type)
; Create a law.
; The law used as part of mathematical calculations
; is NOT saved to the save file.
(define my_law (law "x+x^2-cos(x)"))
;; my_law
; my_law => #[law "(X+X^2)-COS(X)"]
; #[law "(X+X^2)-COS(X)"]
; Evaluate the given law at 1.5 radians
(law:eval my_law 1.5)
;; 3.6792627983323

; Another example of laws.
(define my_edge (edge:circular (position 0 0 0) 20))
;; my_edge
(define my_wirebody (wire-body my_edge))
;; my_wirebody
; The law used as part of the offset is saved to
; the save file.
(define my_offset (wire-body:offset my_wirebody
    "20+10*cos(x*10)"))
;; my_offset
```

light

Scheme Data Type: Lights and Shadows

Description: A light is an entity used for rendering that represents a source of illumination cast on the model. Light types are ambient, distant, eye, point, or spot. Lights can cast shadows if shadow maps are generated. light objects are saved and restored as part of the model.

Derivation: integer : scheme–object

C++ Type: RH_LIGHT

External Rep: #[entity %d %d]
where the first integer is the entity ID, and the second integer is the part ID.

Example:

```
; light (data type)
; Create an inquire a light object.
(define my_light (light "ambient"))
;; my_light
(light? my_light)
;; #t
```

linear–curve

Scheme Data Type: Construction Geometry, Model Geometry

Description: A linear–curve is a data structure describing a curve that is linear in nature. It is used for evaluation purposes and for storing linear–curve data within an entity. linear–curve objects are not saved and restored as part of the model.

Derivation: linear–curve : curve : scheme–object

C++ Type: bounded_line

External Rep: #[curve %x]
where the hexadecimal number is the memory location of the curve. This visually identifies a particular curve, but it cannot be used in Scheme to directly access the curve. Instead, curves should be defined, then the defined name accesses the curve.

Example:
```
; linear–curve (data type)
; Create and inquire a curve:linear object.
(define curve1 (curve:linear (position 0 0 0)
    (position –30 –30 –30)))
;; curve1
(curve:linear? curve1)
;; #t
```

linear–edge

Scheme Data Type: Model Geometry, Model Object

Description: A linear–edge is a topological edge entity that is linear in nature, and that contains a linear–curve data structure describing the edge. linear–edge objects are saved and restored as part of the model.

Derivation: linear–edge : edge : entity : scheme–object

C++ Type: EDGE–>STRAIGHT

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; linear–edge (data type)
; Create and inquire a edge:linear object.
(define edge1
    (edge:linear (position 0 0 0)
    (position 40 40 0)))
;; edge1
(edge:linear? edge1)
;; #t
```

loop

Scheme Data Type: Model Topology

Description: A loop is a topological entity that represents a connected portion of the boundary of a face. Loops are open or closed. A loop can comprise a group of coedges connected in a branched arrangement or in a simple, open chain. A loop can even stand for a coedge shrunk to a single vertex. loop objects are saved and restored as part of the model.

Derivation: loop : entity : scheme–object

C++ Type: LOOP

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; loop (data type)
; Create and inquire a loop object.
(define my_block (solid:block (position 0 0 0)
    (position 23 12 45)))
;; my_block
(define my_loops (entity:loops my_block))
;; my_loops
(loop? (car my_loops))
;; #t
```

lump

Scheme Data Type: Model Topology

Description: A lump is a topological entity that represents a connected 3D (solid) or 2D (sheet) region. A body can contain zero or more lumps. Each lump represents a disjoint set of points. One lump is completely enclosed inside the void of another solid lump. Each lump must have at least one shell. lump objects are saved and restored as part of the model.

Derivation: lump : entity : scheme–object

C++ Type: LUMP

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; lump (data type)
; Create and inquire a lump object.
(define my_block (solid:block (position 0 0 0)
    (position 23 12 45)))
;; my_block
(define my_lumps (entity:lumps my_block))
;; my_lumps
(lump? (car my_lumps))
;; #t
```

material

Scheme Data Type: Materials

Description: A material is an entity used for rendering. Each material specifies a set of color, displacement, reflectance, and transparency shaders that give an object a particular appearance. Materials are created, then attached to zero or more objects. material objects are saved and restored as part of the model.

Derivation: material : entity : scheme–object

C++ Type: RH_MATERIAL

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; material (data type)
; Create and inquire a material object.
(define mat1 (material))
;; mat1
(material? mat1)
;; #t
```

pair

Scheme Data Type: Mathematics

Description: A pair is a scheme–object list of exactly two elements. A pair is constructed using the list procedure. Unlike a list, a pair's external representation consists of two elements, enclosed in parenthesis, separated by a "." character. A list containing two objects is actually two pairs: (object1 . (object2 . void)). Pairs are returned from some extensions. pair objects are not saved and restored as part of the model.

Derivation: pair : scheme–object

C++ Type: None

External Rep: (item1 . item2)
where the "." is a separator character

Example:
```
; pair (data type)
; Create and inquire a pair object.
(define my_pair (list ″thing_1″ ″thing_2″))
;; my_pair
(pair? my_pair)
;; #t
```

part

Scheme Data Type: Part Management

Description: A part is a collection of entities, and is used for grouping, saving, and restoring pieces of a model. A part is not itself an entity. New parts are created, and existing parts are modified, cleaned, or deleted.

Derivation: part : scheme–object

C++ Type: PART

External Rep: #[part %d]
where the number is the part ID number.

Example:
```
; part (data type)
; Create a new part and make it active.
(define my_part1 (part:new))
;; my_part1
(env:set-active-part my_part1)
;; ()
```

par–pos

Scheme Data Type: Mathematics

Description: A par–pos is a parameter space position, consisting of the two coordinates *u* and *v.*

Derivation: par–pos : scheme–object

C++ Type: SPApar_pos

External Rep: #[par–pos %.15g %.15g]
where the first number is the *u* coordinate,
and the second number is the *v* coordinate.

Example:
```
; par–pos (data type)
; Create a par–pos with coordinates u=0.5 and v=0.3
(define pos_1 (par–pos 0.5 0.3))
;; pos_1
```

pattern

Scheme Data Type: Mathematics

Description: A pattern is an object that is used to generate multiple copies of a ”seed” entity and to transform each copy in a way that is characteristic of the particular pattern represented by the object. Lumps, shells, faces, and loops may serve as seed objects and patterns may be represented internally by laws and/or lists of transforms. pattern objects may be saved and restored individually and are automatically saved and restored when attached to an entity (unless doing so would result in a loss of information). In the latter case, patterns are detached from their dependent entities and these entities are saved and restored in their entirety.

Derivation: pattern : scheme–object

C++ Type: pattern

External Rep: #[pattern ”trans_vec” %s ”x_vec” %s ”y_vec” %s ”z_vec” %s ”scale” %s ”keep” %s %x]
where the first five strings define any laws used to generate the pattern and the final string indicates the any uses of a transform list.

Example:

```
; pattern (data type)
; Create, inquire, and apply a pattern object.
(define block (solid:block (position 0 0 0)
    (position 40 40 40)))
;; block
(define a_pattern (pattern:linear
    (gvector 100 0 0) 10))
;; a_pattern
(pattern? a_pattern)
;; #t
(entity:pattern block a_pattern)
;; #[entity 2 1]
(view:compute-extrema dlview)
;; ()
(view:compute-extrema glview)
;; ()
(refresh-all)
;; "refreshed"
```

phl–data

Scheme Data Type: Precise Hidden Line

Description: A phl–data object contains precise hidden line data generated by the phl:compute or phl:retrieve extensions.

Derivation: phl–data : scheme–object

C++ Type: phl_data

External Rep: Not applicable

Example:

```
; phl–data (data type)
; Create a phl–data object.
(define view1 (view:dl))
;; view1
(view:set–eye (position 100 0 100) view1)
;; #[position 0 0 500]
(define b (solid:block (position –10 –25 –35)
    (position 10 25 35)))
;; b
(define c (solid:cylinder (position 0 0 –20)
    (position 0 0 20) 30)))
;; c
(bool:unite c b)
;; [#entity 2 1]
(define phldata1 (phl:compute 1 c view1))
;; phldata1
```

pick–event

Scheme Data Type: Picking

Description: An event object passes position input events into the procedures that compute pick and screen positions. Pick–events are generated from a user mouse click using the read–event extension, or from a set of data using the event procedure.

A pick–event consists of five integers. The first two integers specify the pick's screen *x*–position and *y*–position in pixels. The third integer, ranging from 1 to 3, is the mouse button pressed. The fourth integer is the view ID. The fifth integer is the keyboard state, indicating whether the shift, alt, or control keys were down when the pick occurred. This integer varies between platforms and window managers, and should not be used directly. The extensions event:shift?, event:alt?, and event:control? should be called against the pick–event instead.

Derivation: pick–event : scheme–object

C++ Type: pick_event

External Rep: #[pick–event %d %d %d %d %d]
where the first integer is the *x*–position in screen coordinates,
the second integer is the *y*–position in screen coordinates,
the third integer is the button pressed,
the fourth integer is the view IF,
and the fifth integer is the button state.

Example:

```
; pick–event (data type)
; Create a pick–event object first from a mouse
; click, then from a list of data. Examine one of
; the pick–events.
(define ISO (view:dl))
;; ISO
(define myevent1 (read–event))
;; <click the first mouse button>
;; myevent1
(define myevent2 (event 153 157 1 ISO 0))
;; myevent2
(event? myevent1)
;; #t
(event:button myevent1)
;; 1
(event:left? myevent1)
;; #t
(event:view myevent1)
;; #[view 1075481632]
```

planar–face

Scheme Data Type: Model Geometry, Model Object

Description: A planar–face is a topological entity that is a face that is planar (as opposed to spherical, cylindrical, toroidal, or spline) in nature. A face is a topological entity that is a portion of a single geometric surface. One or more loops of edges bound a face. Faces are open or closed. A face with no loops occupies the entire surface, finite or infinite, on which the face lies. Each face records its sense relative to its underlying surface (same sense or opposite sense). planar–face objects are saved and restored as part of the model.

Derivation: planar–face : face : entity : scheme–object

C++ Type: FACE–>PLANE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; planar–face (data type)
; Create and inquire a planar–face object.
(define my_block (solid:block (position –10 –10 0)
    (position 25 25 25)))
;; my_block
(define my_faces (entity:faces my_block))
;; my_faces
(face:planar? (car (cdr (cdr my_faces))))
;; #t
```

planar–wire

Scheme Data Type: Model Geometry, Model Object

Description: A planar–wire is a topological entity that is a wire that is planar in nature. A wire is a connected collection of edges and/or vertices. Wires typically represent profiles, construction lines, and center lines of swept shapes. Wires can also represent wire frames that, when surfaced, form shells. planar–wire objects are saved and restored as part of the model.

Derivation: planar–wire : wire : entity : scheme–object

C++ Type: WIRE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; planar–wire (data type)
; Create and inquire a planar–wire object.
(define my_one (edge:circular
    (position 0 0 0) 25 0 180))
;; my_one
; Create edge 2.
(define my_two (edge:circular
    (position 0 0 0) 25 180 270))
;; my_two
; Create edge 3.
(define my_three (edge:linear (position 0 0 0)
    (position 25 0 0)))
;; my_three
(define my_wirebody (wire–body (list my_three
    my_one my_two)))
;; my_wirebody
(define my_wires (entity:wires my_wirebody))
;; my_wires
(wire:planar? (car my_wires))
;; #t
```

point

Scheme Data Type: Model Object, Model Topology

Description: A point is a geometric entity (as opposed to a vertex, which is a topological entity) that represents a single point in space. The displayed representation of a point is altered using the env:set–point–size and env:set–point–style extensions. point objects are saved and restored as part of the model.

Derivation: point : entity : scheme–object

C++ Type: APOINT

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; point (data type)
; Create and inquire a point object.
(define my_point (point (position 6.2 6.6 7.9)))
;; my_point
(point? my_point)
;; #t
```

position

Scheme Data Type: Mathematics

Description: A position object represents a location in 3D Cartesian space, and is subject to certain direction and transformation operations. Position coordinates are entered relative to the active coordinate system.

Derivation: position : scheme–object

C++ Type: SPAposition

External Rep: #[position %.15g %.15g %.15g]
where the first double is the *x*–coordinate,
the second double is the *y*–coordinate,
and the third double is the *z*–coordinate.

Example:

```
; position (data type)
; Define and inquire a position object.
(define my_origin (position 0 0 0))
;; my_origin
(position? my_origin)
;; #t
```

ray

Scheme Data Type: Picking, Mathematics

Description: A ray is a composite data type, consisting of a position and a gvector. Rays represent pick direction vectors, infinite lines, or planes. ray objects are not saved as part of the model.

The direction represents a displacement vector in 3D Cartesian space. Internally, a direction is represented in model space coordinates. When coordinates are required, the coordinates are entered relative to the active coordinate system.

Derivation: ray : position : gvector : scheme–object

C++ Type: pick_ray

External Rep: #[ray (%g %g %g) (%g %g %g)]
where the first triplet of doubles is the (x y z) position, and the second triplet of doubles is the (x y z) gvector.

Example:

```
; ray (data type)
; Define and inquire a ray object.
(define x-ray (ray (position 0 0 0) (gvector 1 0 0)))
;; x-ray
(ray? x-ray)
;; #t
```

rbd–driver

Scheme Data Type: Rubberbanding

Description: An rbd–driver is a scheme–object that defines a rubberband driver. A C++ rubberband_driver is a set of callback functions packaged as a class with virtual functions. There is a virtual function for each rubberbanding event: start, update, stop, and repaint.

Derivation: rbd–driver : scheme–object

C++ Type: rubberband_driver

External Rep: #[rbd–driver %x]
where the hex value is the pointer to the object.

Example:
```
; rbd–driver (data type)
; Create a line rubberband driver and make it active.
(define my_rbd (rbd:line #t (position 1 2 3)))
;; my_rbd
```

rbd–scheme–driver

Scheme Data Type: Rubberbanding

Description: An rbd–scheme–driver is a scheme–object that holds a rubberband driver. In addition to the driver, it holds the mouse hooks that attach Scheme mouse procedures to the drivers. Rubberbanding hooks are saved as part of the rubberband_scheme class as a vector of procedures. Rubberband_scheme also keeps a vector of arbitrary scheme–objects, which the driver can use to keep any local data.

Derivation: rbd–scheme–driver : scheme–object

C++ Type: rubberband_scheme

External Rep: #[rbd–scheme–driver %x]
where the hex value is the pointer to the object.

Example:
```
; rbd–scheme–driver (data type)
; Create a scheme rubberband driver and
; make it active.
; Create a vector to hold 7 elements
(define the_rb_hooks (make–vector 7))
;; the_rb_hooks
; Called once when the driver is activated.
; Can be used to set globals or the elements
; in the locals vector.
(define the_init_hook (lambda (self)
    (display ”rubberband driver: INIT.\n”)))
;; the_init_hook
```

```
; Called when driver activated and when mouse enters
; a window.
(define the_start_hook (lambda (self pick_event)
    (display "rubberband driver: START.\n")
    (rbd:scheme-set-local self 0
        (pick:position pick_event))
    (rbd:scheme-get-position self pick_event)))
;; the_start_hook
; Called when the mouse moves within a window.
(define the_update_hook (lambda (self pick-event)
    (display "rubberband driver: UPDATE.\n")
    (rbd:scheme-set-local self 1
        (pick:position pick-event))))
;; the_update_hook

; Called when mouse leaves window and when driver is
; deactivated.
(define the_stop_hook (lambda (self)
    (display "rubberband driver: STOP.\n")))
;; the_stop_hook

; Called when view receives repaint event.
(define the_repaint_hook (lambda (self view)
    (display "rubberband driver: REPAINT.\n")))
;; the_repaint_hook

; May be called by other hooks to map a pick-event to
; a position.
(define the_position_hook (lambda (self pick-event)
    (display "rubberband driver: POSITION.\n")
    (display "position not changed\n")))
;; the_position_hook

; Called once when the driver is deactivated.
(define the_end_hook (lambda (self)
    (display "rubberband driver: END.\n")))
;; the_end_hook
(vector-set! the_rb_hooks 0 the_init_hook)
(vector-set! the_rb_hooks 1 the_start_hook)
(vector-set! the_rb_hooks 2 the_update_hook)
(vector-set! the_rb_hooks 3 the_stop_hook)
(vector-set! the_rb_hooks 4 the_repaint_hook)
(vector-set! the_rb_hooks 5 the_position_hook)
(vector-set! the_rb_hooks 6 the_end_hook)
;; ()
```

```
; Non-scheme rbd drivers can use the global variable
; 'rb-position-hook', whereas, Scheme rbd drivers use
; either that or 'the_position_hook'

; ----------------------------------
; start rubberbanding
; ----------------------------------
; Create and initialize some local variables
(define the_locals (make-vector 2))
(vector-set! the_locals 0 (position 0 0 0))
(vector-set! the_locals 1 (position 0 0 0))
;; ()

; Create the Scheme rubberband driver
(define the_scm_rbd
    (rbd:scheme #f the_rb_hooks the_locals))
;; the_scm_rbd

the_scm_rbd
;; #[rbd-scheme-driver 4020fce0]

; Begin rubberbanding
(read-event)
;; #[pick-event 185 455 1 1075924776 0]

(rbd:push the_scm_rbd)
;; rubberband driver: INIT.
;; (#[rbd-driver 4021eb48])

(read-event)
;; rubberband driver: START.
;; rubberband driver: POSITION.
;; position not changed
;; rubberband driver: UPDATE.
;; ...
;; rubberband driver: UPDATE.
;; rubberband driver: UPDATE.
;; rubberband driver: STOP.

(rbd:pop)
;; rubberband driver: END.
;; (#[rbd-driver 4021eb48])

; display results
(display "Start Position: ")
    (display (vector-ref the_locals 0))
    (newline)
;; Start Position:
;; #[position 38.1656983627057 -92.3988426777293 0]
```

```
(display ”Stop Position: ”)
    (display (vector–ref the_locals 1))
    (newline)
;; Stop Position:
;; #[position 28.72230 –96.44886 –7.105e–15]
```

real

Scheme Data Type: Mathematics

Description: A real object is a Scheme language primitive containing a single real (double) value. An integer is always also a real. Not all reals are integers. real objects are not saved as part of the model unless they are part of an entity.

Derivation: real : scheme–object

C++ Type: double

External Rep: %g

Example:

```
; real (data type)
; Define and inquire a real object.
(define pi 3.14159265359)
;; pi
(real? pi)
;; #t
```

refinement

Scheme Data Type: Faceting, Modeler Control

Description: A refinement is an entity that defines how faceting is applied to a geometric object. Refinements are created, their parameters set, then are applied to geometric entities. refinement objects are saved and restored as part of the model.

Derivation: refinement : entity : scheme–object

C++ Type: REFINEMENT

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; refinement (data type)
; Create and inquire a refinement object.
(define my_cyl (solid:cylinder (position 0 0 0)
    (position 30 30 0) 20))
;; my_cyl
(define my_ref (refinement))
;; my_ref
(refinement:set–prop my_ref
    ”aspect ratio” 0.5)
;; ()
(entity:set–refinement my_cyl my_ref)
;; #[entity 2 1]
(entity:refinement my_cyl)
;; #[entity 3 1]
(refinement? my_ref)
;; #t
```

scm_cvty

Scheme Data Type: Scheme AIDE Application, Scheme Interface, Mathematics

Description: Represents the convexity at a point or along a single edge (or something equivalent), such as convex, tangent convex, etc. For more information about the possible convexity values, refer to files pt_cvty.hxx and ed_cvty.hxx. A scm_cvty is not created directly; it is created by instantiating either a scm_pt_cvty_info or scm_ed_cvty_info.

Derivation: scm_cvty : scheme–object

C++ Type: cvty

External Rep: #[cvty: “%s”]
where the quoted string(s) represents the convexity. One or more convexity strings may be returned, where each individual string indicates some property of the convexity. For example, if both “tgt” and “cvx” are returned, the convexity is “tangent convex.” The possible convexity strings are:

cvx	= Convex
cve	= Concave
tgt	= Tangent
infl	= Inflection
knf	= Knife
mxd	= Mixed
unk	= Unknown
unset	= Unset

Example:

```
; scm_cvty
; Create a block.
(define block1 (solid:block (position 0 10 0)
    (position 10 20 20)))
;; block1
; Define the edges of block1
(define edge–list (entity:edges block1))
;; edge–list
; Instantiate to create a scm_cvty
(ed–cvty–info:instantiate (edge:ed–cvty–info
    (list–ref edge–list 0)) .01)
; Convexity is convex
;; #[cvty: cvx]
```

scm_ed_cvty_info

Scheme Data Type: Scheme AIDE Application, Scheme Interface, Mathematics

Description: Represents the convexity of an edge (or equivalent). Refer to scm_cvty for more information.

Derivation: scm_ed_cvty_info : scheme–object

C++ Type: ed_cvty_info

External Rep: #[ed_cvty_info: [%g, %g] [cvty: “%s”]]
where the doubles are the maximum and minimum angles between surface normals along this edge (positive indicates convex, negative indicates concave); the quoted string(s) is the convexity of the edge, assuming an angle tolerance such that the whole edge would be regarded as tangent (refer to scm_cvty for the convexity strings).

Example:

```
; scm_ed_cvty_info
; Create a block.
(define block1 (solid:block (position 0 10 0)
    (position 10 20 20)))
;; block1
; Define the edges of block1
(define edge–list (entity:edges block1))
;; edge–list
(edge:ed–cvty–info (list–ref edge–list 0))
;; #[ed_cvty_info: [1, 1] [cvty: knf]]
```

scm_pt_cvty_info

Scheme Data Type: Scheme AIDE Application, Scheme Interface, Mathematics

Description: Represents the convexity of a single point along an edge (or equivalent). Refer to scm_cvty for more information.

Derivation: scm_pt_cvty_info : scheme–object

C++ Type: pt_cvty_info

External Rep: #[pt_cvty_info: %g %s %g
where the first double is the angle between surface normals at this point (positive indicates convex, negative indicates concave); the quoted string(s) is the convexity at the point, assuming an angle tolerance such that the point would be regarded as tangent (refer to scm_cvty for the convexity strings); the second double is the default tolerance, which is a value derived from the surface curvatures at this point.

Example:
```
; scm_pt_cvty_info
; Define some geometry and return the convexity.
(define w (solid:wiggle 60 60 60 "sym"))
;; w
(define edge (list-ref (entity:edges w) 0))
;; edge
(edge:mid-pt-cvty-info edge)
;; #[pt_cvty_info: 0.72297780254992 [cvty: cvx knf]
;; (tol 0.00019906840640487)]
```

scheme–object

Scheme Data Type: Scheme AIDE Application, Scheme Interface, Mathematics

Description: A scheme–object is the highest level container data type. All other data types and procedures are scheme–objects. All data type inquiry procedures (procedures with names ending in "?") take any scheme–object as their argument.

Derivation: scheme–object

C++ Type: None

External Rep: Not applicable

Example:
```
; scheme-object (data type)
; Define a scheme-object that is a position
(define p (position 10 10 10))
;; p
```

scheme–procedure

Scheme Data Type: Scheme AIDE Application, Scheme Interface

Description: A scheme–procedure is a scheme–object that contains a procedure that is executed by the interpreter. Procedures are defined using the lambda operator. scheme–procedure objects are not saved and restored as part of the model.

Derivation: scheme–procedure: scheme–object

C++ Type: None

External Rep: #[compound %x]
where the hex number is a pointer to the procedure.

Example:
```
; scheme–procedure (data type)
; Create a procedure object.
(define my_render (lambda (x) render x))
;; my_render
```

section

Scheme Data Type: Skinning and Lofting

Description: The section Scheme data type is a data structure used as input to the sheet:loft–wires extension.

```
(section my_coedges in_flag take_off_factor
    [no_loop_flag])
```

– no_loop_flag boolean
– in_flag boolean
– my_coedges coedge | coedge ...
– take_off_factor real

The my_coedges argument is a list of one or more coedges to be used as one section of the loft operation. Only one coedge of a loop has to be specified when the argument no_loop_flag is set to #f; all other coedges connected with the given my_coedges in one or more loops are added to the list.

The argument in_flag specifies whether the take–off vector is coming into (#t) or out of (#f) the surface at the given coedge. Typically, the first section has its coedges labeled as out of (#f) and all others sections created for the loft operation label their coedges in the list as into (#t).

The argument take_off_factor specifies the magnitude of the take–off vector as it leaves the given coedge. The lofted surface is always tangent to the surface bounded by the my_coedges coedges.

The take_off_factor is applied to the magnitude of the take–off vector, and can be used to control the shape of the loft surface. The take–off vector is a tangent vector going out of a given surface and into the lofted surface. The lofted surface is always tangent to the surface bounded by the coedges. Small values for the weighting of the take–off vector mean that the transition from the tangent to the lofted surface happens abruptly. Large values for the weighting of the take–off vector mean that the transition from the tangent to the lofted surface happens more gradually. Extremely high weight values could result in excessive whipping in the lofted surface, if not a self–intersecting surface.

The option argument no_loop_flag is set to #f by default. This means that only one coedge of a loop has to be specified; all other coedges connected with the given coedge are automatically added to the list. When the no_loop_flag is set to #t, it means that only the specified coedges in my_coedges are to be used as that part of the section; no connecting loop coedges are added to the list.

Derivation: section : entity : scheme–object

C++ Type: None

External Rep: #[section “number of coedges = %d, sense, %d]
where the first integer specifies the number of coedges in the section, sense specifies whether the lofted surface is out of (FORWARD) or into (REVERSE) the coedge, and the second integer specifies the magnitude of the take–off vector.

Example:

```
; section
; Establish the correct options for viewing.
(option:set ”cone_par” #t)
;; #f
; Turn off silhouettes for faster
; calculation.
(option:set ”sil” #f)
;; #t
; Turn on parameter lines.
(option:set ”u_par” 5 )
;; –1
(option:set ”v_par” 7 )
;; –1
; Create a cylindrical face.
(define my_face1 (face:cylinder (position 0 0 0)
    (position 0 20 0) 20 ))
;; my_face1
```

```
; Create a second cylindrical face.
(define my_face2 (face:cylinder (position 0 40 0)
    (position 0 60 0) 50 ))
;; my_face2
; Get the coedge of the first face.
(define my_coedges1 (entity:coedges my_face1))
;; my_coedges1
; Get the coedges of the second face.
(define my_coedges2 (entity:coedges my_face2))
;; my_coedges2
; Define a section with a large take–off vector.
(define my_section1 (section
    (list (list–ref my_coedges1 1)) #f 10))
;; my_section1
; my_section1 =>
; #[section ”number of coedges = 1,
;       FORWARD, 10.000000”]
; Define second section with a small take–off vector.
; and reverse direction.
(define my_section2 (section
    (list (list–ref my_coedges2 0)) #t 1))
;; my_section2
; my_section2 =>
; #[section ”number of coedges = 1,
;   REVERSE, 1.000000”]
; Create a lofted sheet body.
(define my_sheet (sheet:loft–wires
    (list my_section1 my_section2) #f #t #t))
;; my_sheet
; Color created loft face to view better.
(entity:set–color my_sheet 6)
;; ()
; To view this from different angles.
; (load ”rotsph.scm”)
; (rotsph #t)
; Move with mouse.
```

shell

Scheme Data Type: Model Topology

Description: A shell is a topological entity consisting of a set of connected faces. The faces are connected along either edges or vertices. The shell represents a sheet region, bounds a solid region, or both. A shell that bounds a solid region is entirely peripheral, or void, or neither. shell objects are saved and restored as part of the model.

Derivation: shell : entity : scheme–object

C++ Type: SHELL

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; shell (data type)
; Create and inquire a shell object.
(define my_block (solid:block (position 0 0 0)
    (position 8 8 8)))
;; my_block
(define my_shells (entity:shells my_block))
;; my_shells
(shell? (car my_shells))
;; #t
```

skin_options

Scheme Data Type: Skinning and Lofting

Description: A skin_options is a Scheme data type that holds options for some skinning and lofting Scheme extensions.

This data type is created by the skin:options Scheme extension. All values are set as UNSET (–1) as default and every API called will set its own default value if UNSET. Options remain unchanged after being applied to Scheme extensions, so they can be used in other functions with the security that it is the same set of options defined. Refer to the skin:options Scheme extension or the skin_options class for more details.

Derivation: skin_options : scheme–object

C++ Type: skin_options

External Rep: #[Skin_Options ”arc_length” %b ”no_twist” %b ”align” %b ”perpendicular” %b ”simplify” %b ”closed” %b ”solid” %b ”periodic” %b ”virtualGuides” %b]

Example:

```
; skin_options (data type)
; Set up the view.
(option:set ”match_paren” #f)
;; #t
(view:delete)
;; ()
(define glview (view:gl 0 0 300 300))
;; glview
(define edgeview (view:edges #t))
;; edgeview
(define polyview (view:polygonoffset #t))
;; polyview
(define verticesview (view:vertices #t))
;; verticesview
(part:clear)
;; #t
; Define start and end wires.
(define wire–1 (wire–body (list (edge:ellipse
    (position 0 0 0) (gvector 0 0 1)
    (gvector 0 5 0) 1 0 180)
    (edge:ellipse (position 0 0 0)
    (gvector 0 0 1) (gvector 0 5 0) 1 180 360))))
;; wire–1
(define wire–2 (wire–body (list (edge:ellipse
    (position 0 0 70) (gvector 0 0 1)
    (gvector 0 5 0) 1 0 180)
    (edge:ellipse (position 0 0 70)
    (gvector 0 0 1) (gvector 0 5 0) 1 180 360))))
;; wire–2
(define myWires (list wire–1 wire–2))
;; myWires
; Create spline edge.
(define guide (edge:spline (list (position 0 5 0)
    (position 0 5 8) (position 0 6 16)
    (position 0 3 24) (position 0 6 32)
    (position 0 3 40) (position 0 6 48)
    (position 0 3 54) (position 0 9 62)
    (position 0 5 70))))
(define viewset (view:set (position 155 –100 340)
    (position 0 4 0) (gvector –1 0 .5)))
;; viewset
(define zoom (zoom–all))
;; zoom
; Define skin_options.
(define opts (skin:options ”arc_length” #f ”no_twist”
```

```
    #t "align" #f "simplify" #f "closed" #f
    "solid"#t "virtualGuides" #t))
;; opts
(define myBody
    (sheet:skin–wires–guides myWires guide opts))
; Roll back and redefine options.
(roll)
;; –1
(define skin (skin:options "solid" #f "virtualGuides"
    #f opts))
;; skin
(define myBody (sheet:skin–wires–guides myWires guide
    opts))
;; myBody
```

SLInterface

Scheme Data Type: Skinning and Lofting

Description: The SLInterface Scheme data type is a data structure used to control skinning and lofting operations.

Derivation: SLInterface : scheme–object

C++ Type: AcisSLInterface

External Rep: #[skinning interface object]

Example:
```
; SLInterface
; Example not available at this time.
```

spherical–face

Scheme Data Type: Model Object, Model Geometry

Description: A spherical–face is a topological entity that is a face that is spherical in nature. A face is a topological entity that is a portion of a single geometric surface. A face records its sense relative to its underlying surface (same sense or opposite sense). spherical–face objects are saved and restored as part of the model.

Derivation: spherical–face : face : entity : scheme–object

C++ Type: FACE–>SPHERE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; spherical–face (data type)
; Create and inquire a spherical–face object.
(define my_sph (solid:sphere (position 0 0 0) 20))
;; my_sph
(define my_faces (entity:faces my_sph))
;; my_faces
(face:spherical? (car my_faces))
;; #t
```

splgrid

Scheme Data Type: Construction Geometry, Spline Interface

Description: A splgrid is a lightweight data structure that holds a description of a spline surface. It specifies the information using a grid of positions on the surface. A splgrid is not an entity and is not saved and restored as part of the model. An splgrid is converted into an entity using the face:spline–grid command.

Derivation: splgrid : scheme–object

C++ Type: splgrid

External Rep: #[splgrid %x]
where the hex number specifies the object's memory location.

Example:

```
; splgrid (data type)
; Create a spline surface grid.
(splgrid)
;; #[splgrid bbf918]
```

splsurf

Scheme Data Type: Construction Geometry, Spline Interface

Description: A splsurf is a lightweight data structure that holds a description of a spline surface. It specifies the information using a set of control points, knots, and weights. A splsurf is not an entity and is not saved and restored as part of the model. An splsurf is converted to an entity using the face:spline–ctrlpts command.

Derivation: splsurf : scheme–object

C++ Type: splsurf

External Rep: #[splsurf %x]
where the hex number specifies the object's memory location.

Example:

```
; splsurf (data type)
; Create a spline surface.
(splsurf)
;; #[splsurf bbf9C0]
```

spline–edge

Scheme Data Type: Model Object, Model Geometry, Spline Interface

Description: A spline–edge is a topological entity that is an edge represented by a spline curve. An edge is a topological entity associated with a curve. An edge is bounded by one or more vertices, and refers to one vertex at each end. If the reference at either or both ends is NULL, the edge is unbounded in that direction. Each edge contains a record of its sense (FORWARD or REVERSED) relative to its underlying curve. spline–edge objects are saved and restored as part of the model.

Derivation: spline–edge : edge : entity : scheme–object

C++ Type: EDGE–>INTCURVE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; spline-edge (data type)
; Create and inquire a spline-edge object.
(define my_edge (edge:spline
    (list (position 0 0 0) (position 5 5 0)
    (position 10 15 0) (position 15 20 0)
    (position 20 15 0) (position 25 5 0)
    (position 30 0 0)) 0 45))
;; my_edge
(edge:spline? my_edge)
;; #t
```

spline–face

Scheme Data Type: Model Object, Model Geometry, Spline Interface

Description: A spline–face is a topological entity that is a face that is represented by a spline function. A face is a topological entity that is a portion of a single geometric surface. One or more loops of edges bound a face. Faces are open or closed. A face with no loops occupies the entire surface, finite or infinite, on which the face lies. Thus a face may stand for an infinite plane or for a complete sphere. Each face records its sense relative to its underlying surface (same sense or opposite sense). spline–face objects are saved and restored as part of the model.

Derivation: spline–face : face : entity : scheme–object

C++ Type: FACE–>SPLINE

External Rep: #[entity %d %d]
where the first integer is the entity ID, and the second integer is the part ID.

Example:
```
; spline–face (data type)
; Create and test a spline–face.
(define e1 (edge:spline (list (position 0 0 0)
    (position 20 –20 0) (position 40 0 0))))
;; e1
(define e2 (edge:linear (position 40 0 0)
    (position 40 40 0)))
;; e2
(define e3 (edge:linear (position 40 40 0)
    (position 0 40 0)))
;; e3
(define e4 (edge:linear (position 0 40 0)
    (position 0 0 0)))
;; e4
(define w (wire–body (list e1 e2 e3 e4)))
;; w
(define my_body (solid:sweep–wire w
    (gvector 0 0 40)))
;; my_body
(define my_faces (entity:faces my_body))
;; my_faces
(face:spline? (car (cdr (cdr (cdr my_faces)))))
;; #t
```

string

Scheme Data Type: Mathematics, Text

Description: A string is a scheme–object containing a text string. Strings often display messages to the output window. string objects are not saved and restored as part of the model.

Derivation: string : scheme–object

C++ Type: char*

External Rep: ”%s”

Example:

```
;; string (data type)
;; Define and inquire a string object.
(define testing ”This is a test.”)
;; testing
(string? testing)
;; #t
(string–length testing)
;; 15
```

surface

Scheme Data Type: Model Geometry, Construction Geometry

Description: The surface object is the base from which specific surface types (plane, cone, sphere, torus, and spline) are derived. surface objects are not saved and restored as part of the model.

All surfaces have a parameterization scheme defined for them. However, the analytic surfaces (plane, cone, sphere, and torus) are not considered parametric surfaces. The only “true” parametric surface in ACIS is the spline surface.

The parameterization of a surface maps a rectangle within a 2D vector space (*u*,*v* parameter space) into a 3D real vector space *(x*,*y*,*z* object space). A surface is closed in *u* (or *v*) if the opposite sides of the rectangle map into identical curves in object space. If the derivatives also match at these boundaries, the surface is periodic in that parameter. If one side of this rectangle maps into a single point in object space, this point is a parametric singularity. If the surface normal is not continuous at this point, the point is a surface singularity.

The parameterization is either right–handed, i.e., the surface normal is the cross product of *u* and *v*, or left–handed, i.e., the normal is the cross product of *v* and *u*.

Derivation: surface : scheme–object

C++ Type: surface

External Rep: #[%s surface, %x] or #[surface, %x]
where the string is the surface type name,
and the hexadecimal number is the memory location of the surface object.

Example:

```
; surface (data type)
; Create and inquire a surface object.
(define my_block (solid:block
    (position 0 0 0) (position 20 30 40)))
;; my_block
(define my_faces (entity:faces my_block))
;; my_faces
(surface:from-face (car (cdr my_faces)))
;; #[plane surface %x]
```

Sweep_Options

Scheme Data Type: Laws, Sweeping

Description: A Sweep_Options is a Scheme data type that holds options for the sweep Scheme extension. This data type is created by the sweep:options Scheme extension. Not all of the options appear in the list. They only appear if they are explicitly specified in the sweep:options Scheme extension.

Some of the options are mutually exclusive. The option “draft_angle” is mutually exclusive with “draft_law”, which is mutually exclusive with “start_draft_dist” / “end_draft_dist” Internally, all of these options are converted to the equivalent law.

The option “twist_angle” is mutually exclusive with “twist_law”. Internally, these options are converted to the equivalent law. Refer to the sweep:options Scheme extension or the sweep_options class for more details.

Derivation: Sweep_Options : scheme–object

C++ Type: sweep_options

External Rep: #[Sweep_Options “solid” %b {[“draft_angle” %lf] | [“draft_law” %s] |
[“draft_start_distance” %lf “draft_end_distance” %lf”]}
“gap_type” %s {[“twist_angle” #lf] | [“twist_law” %s]}
[“to_face” %x] [“rail_law” %s] [“scale_law” %s”].

Example:

```
; Sweep_Options (data type)
; Define the sweep options to use.
; These are the default options.
(define my_sweep_default (sweep:options))
;; my_sweep_default
; my_sweep_default =>
; #[Sweep_Options ”solid” #t ”draft_angle” 0.000000
; ”gap_type” 0 ”twist_angle” 0.000000]
; Define new sweep options where all values are
; default except the draft_law.
(define my_sweep_s1 (sweep:options
    ”draft_law” ”sin(x)”))
;; my_sweep_s1
; my_sweep_s1 =>
; #[Sweep_Options ”solid” #t ”draft_angle” 0.000000
; ”draft_law” SIN(X) ”gap_type” 0
; ”twist_angle” 0.000000]
; Define another set of sweep options that is almost
; the same as my_sweep_s1 except for a minor change
; to the gap_type to be ”natural”.
(define my_sweep_s2 (sweep:options
    ”gap_type” ”n” my_sweep_s1))
;; my_sweep_s2
; my_sweep_s2 =>
; #[Sweep_Options ”solid” #t ”draft_angle” 0.000000
; ”draft_law” SIN(X) ”gap_type” 2
; ”twist_angle” 0.000000]
```

```
; Another Example.
; Create a sweep path from points
(define my_plist (list (position 0 0 0)
    (position 20 0 0) (position 20 20 0)
    (position 20 20 20)))
;; my_plist
(define my_start (gvector 1 0 0))
;; my_start
(define my_end (gvector 0 0 10))
;; my_end
(define my_path (edge:spline my_plist
    my_start my_end))
;; my_path
(define my_law (law "cur(edge1)" my_path))
;; my_law
(define my_rail (law "minrot(law1,vec(0,–1,0))"
    my_law))
;; my_rail
(define my_edge1 (edge:linear (position 0 3 3)
    (position 0 3 –3)))
;; my_edge1
(define my_edge2 (edge:linear (position 0 3 –3)
    (position 0 –3 –3)))
;; my_edge2
(define my_edge3 (edge:linear (position 0 –3 –3)
    (position 0 –3 3)))
;; my_edge3
(define my_edge4 (edge:linear (position 0 –3 3)
    (position 0 3 3)))
;; my_edge4
(define my_profile (wire–body
    (list my_edge1 my_edge2 my_edge3 my_edge4)))
;; my_profile
(define my_sweep (sweep:law my_profile my_path
    (sweep:options "rail_law" my_rail)))
;; my_sweep
```

tcoedge

Scheme Data Type: Tolerant Modeling, Healing

Description: A tcoedge or “tolerant coedge” is a coedge derived from the COEDGE class that includes a tolerance value. The tolerance coedge class, TCOEDGE, is a derived class of COEDGE. The TCOEDGE extends the COEDGE class by adding parameter bounds, a lazy 3D curve derived from the pcurve, and a parameter box. TCOEDGEs on faces must maintain a pointer to a pcurve, which represents the curve underlying the edge in the parametric space of the surface. TCOEDGEs on analytic surfaces are required to have pcurves.

Derivation: tcoedge : coedge : entity : scheme–object

C++ Type: TCOEDGE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; tcoedge (data type)
; Create something with tolerant topology
```

tedge

Scheme Data Type: Tolerant Modeling, Healing

Description: The tolerant edge class is a derived class of EDGE with a tolerance value. Unlike an edge, whose 3D representation is provided by its underlying curve, the 3D representation of a TEDGE is defined by the 3D geometry of its associated TCOEDGEs. The curve underlying a tolerant edge is used purely for graphic visualization.

The tolerance value of a tolerant edge indicates the maximum distance between any two equiparametric positions on any of its tolerant coedges. Two tolerant edges are coincident over an interval if the maximum of the minimum distance between the portion of their point sets bounded by the interval is less than the maximum of the tolerant edge’s tolerance values. If the tolerance value is less than SPAresabs, then SPAresabs is used for coincidence checking.

A tolerant edge and an edge are coincident over an interval if the maximum of the minimum distance between the portion of their point sets bounded by the interval is less than the maximum of the tolerant edge’s tolerance value and SPAresabs.

A tolerant edge with a single tolerant coedge will have a zero tolerance value. A tolerant edge may only be associated with tolerant coedges.

Derivation: tedge : edge : entity : scheme–object

C++ Type: TEDGE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; tedge (data type)
; Create something with tolerant topology
(define block1 (solid:block
    (position 0 0 0) (position 50 50 50)))
;; block1
(define edge1 (car (entity:edges block1)))
;; edge1
(define tol_edge (edge:tolerant edge1))
;; tol_edge
```

text

Scheme Data Type: Text

Description: A text is an entity that annotates the model in graphics windows and is always displayed parallel to the viewing plane. Transforming a text object's position of origin moves the text, but it does not change its real or perceived size. text objects are saved and restored as part of the model.

A text object is a composite of a string, a position, a font, and a size. The position specifies the location of the string's first character (left edge, baseline the string sits on). Font specifies the typeface used. Size specifies the displayed size in points. Refer to the text extension for more information.

Derivation: text : entity : scheme–object

C++ Type: TEXT_ENT

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; text (data type)
; Define and inquire a text object.
(define hi (text (position 0 0 0) "Hello world."
    "times-medium-r-normal" 30))
;; hi
(text? hi)
;; #t
(text:font hi)
;; "times-medium-r-normal"
(text:size hi)
;; 30
(text:string hi)
;; "Hello world."
```

texture–space

Scheme Data Type: Texture Spaces

Description: A texture–space is an entity used during rendering to alter the shading transforms to make an object appear to be carved out of a material, or to have a material wrapped around the object. Texture space types include “x”, “y”, “z”, “cylindrical”, “spherical”, and “auto–axis”. Display of the effects of texture spaces require the Advanced Rendering Component. texture–space objects are saved and restored as part of the model.

Derivation: texture–space : entity : scheme–object

C++ Type: RH_TEXTURE_SPACE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; texture-space (data type)
; Create and inquire a texture space object.
(define my_space (texture-space "cylindrical"))
;; my_space
(texture-space? my_space)
;; #t
```

tm–chk–info

Scheme Data Type: Tolerant Modeling

Description: A tm–chk–info

Derivation: tm–chk–info : scheme–object

C++ Type: tm_chk_info

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; tm–chk–info (data type)
; Example not available at this time.
```

toroidal–face

Scheme Data Type: Model Object, Model Geometry

Description: A toroidal–face is a topological entity that is a face that is part or all of a torus. Each face records its sense relative to its underlying surface (same sense or opposite sense). toroidal–face objects are saved and restored as part of the model.

Derivation: toroidal–face : face : entity : scheme–object

C++ Type: FACE–>TORUS

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; toroidal–face (data type)
; Create and inquire a toroidal–face.
(define my_torus (solid:torus
    (position –10 –10 –10) 7 3))
;; my_torus
(define my_faces (entity:faces my_torus))
;; my_faces
(face:toroidal? (car my_faces))
;; #t
```

transform

Scheme Data Type: Transforms

Description: A transform is an object that translates, rotates, scales, and reflects top–level entities. Top level entities are defined as entities that do not make up part of another containing entity. For example, bodies and WCSs are top level entities. Lumps, loops, edges, coedges, faces, shells, wires, and vertices contained within a body are not top level entities. transform objects are not saved and restored as part of the model.

Derivation: transform : scheme–object

C++ Type: SPAtransf

External Rep: #[transform %u]
where the unsigned integer is the memory location of the transform.

Example:
```
; transform (data type)
; Create, inquire and apply a transform object.
(define my_block (solid:block
    (position 0 0 0) (position 40 40 40)))
;; my_block
(define mytransf (transform:axes (position 0 0 0)
    (gvector 10 0 0) (gvector 12 12 0)))
;; mytransf
(transform? mytransf)
;; #t
(entity:transform my_block mytransf)
;; #[entity 2 1]
```

tube_options

Scheme Data Type: Booleans

Description: A tube_options is a Scheme data type that holds options for the tube_options Scheme extension. This data type is created by the tube:options Scheme extension.

The purpose of this data structure is to hold information between selective Boolean operations stages 1 and 2 and to make selective Booleans accessible by other modeling operations.

Derivation: integer : scheme–object

C++ Type: tube_options

External Rep: #[tube_options “keep_law” %s “keep_branches” %b
{[“limit”] | [“unite”] | [“intersect”] | [“subtract”]}.

Example:

```
; tube_options (data type)
; Create tube_options in Boolean
(define b (solid:block (position 0 0 0)
    (position 10 10 10)))
;; b
; b => #[entity 2 1]
(define c1 (solid:cylinder (position –5 5 2)
    (position 15 5 2)1))
;; c1
(define c2 (solid:cylinder (position –5 5 8)
    (position 15 5 8)1))
;; c2
(define c(solid:unite c1 c2))
;; c
(define start (list (list–ref (entity:faces c) 1)
    (list–ref (entity:faces c) 4)))
;; start
(define end (list (list–ref (entity:faces c) 2)
    (list–ref (entity:faces c) 5)))
;; end
(define opt1 (tube:options ”keep_law” ”x=0”
     ”bool_type” ”UNITE”))
;; opt1
opt1
;; #[tube_options ”keep_law” <null>
    ”keep_branches” #f”unite”]
(define d(bool:tube b c start end opt1))
;; d
```

```
; Another Example.
(part:clear)
;; #t
(define b (solid:block (position 0 0 0)
    (position 10 10 10)))
;; b
(define c (solid:cylinder (position –5 5 5)
    (position 15 5 5)2))
;; c
(define start (list–ref (entity:faces c) 1))
;; start
(define end (list–ref (entity:faces c) 2))
;; end
(define opt1 (tube:options ”keep_law”
    ”x=0” ”bool_type” ”unite”))
;; opt1
; opt1 =>#
; [Tube_Options ”keep_law” X=0 ”keep_branches”
    #f”unite”]
(define d (bool:tube b c start end opt1))
;; d
```

tvertex

Scheme Data Type: Tolerant Modeling, Healing

Description: A tvertex or “tolerant vertex” is a vertex derived from the VERTEX class that includes a tolerance value.

The tolerant vertex class, TVERTEX, is a derived class of VERTEX with a tolerance value. A tolerant vertex is a 0–dimensional, topological entity that is used to bound an edge or a tedge. Each tolerant vertex is represented by a point in the geometric model. The tolerance value indicates the maximum distance from the position of the tvertex to the end of each coedge.

Two tolerant vertices are coincident if the distance between their points is less than the maximum of their tolerance values. If the tolerance value is less than SPAresabs, then SPAresabs is used for coincidence checking.

A tolerant vertex and a vertex are coincident if the distance between their points is less than the maximum of the tolerant vertex tolerance value and SPAresabs.

Derivation: tvertex : vertex : entity : scheme–object

C++ Type: TVERTEX

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:
```
; tvertex (data type)
; Create something with tolerant topology
```

unspecified

Scheme Data Type: Mathematics

Description: The return type for some Scheme extensions is unspecified. The actual type and content of the return could be anything, but is usually an empty list. Such returns should be ignored and not passed into other procedures.

Derivation: None

C++ Type: None

External Rep: Not applicable

Example:

```
; unspecified (data type)
; Demonstrate an unspecified return.
(define my_part1 ( part:new))
;; my_part1
(history my_part1)
;; #[history 0 2]
(roll:debug (history my_part1) 2)
; ------------------------------------------------
; HISTORY_STREAM :4011af40
; attribute = 0
; active_ds = 2
; current_ds = 0
; root_ds = 2
; current_state = 2
; link_states = TRUE
; next_state = 2
; ------------------------------------------------
; Delta state this 2, 4011af40
; : this 2 backward 1
; next_ds –1,0
; prev_ds –1,0
; partner_ds 2, 4011af40
; owner_stream 4011af40
;       user_data 0
;       name NULL
; No bulletin boards
;; ()
; This function returns an unspecified type.
```

vector

Scheme Data Type: Scheme Interface

Description: A vector is a native Scheme data type that corresponds to an array in C++. Vectors are heterogeneous structures whose elements are indexed by integers. The length of a vector is the number of elements it contains, and is fixed when the vector is created. Indexes into a vector are integers between 0 and the length of the vector. vector objects are not saved and restored as part of the model. The ACIS–defined Scheme data type gvector is used to represent a (mathematical) vector with magnitude and direction.

Derivation: vector : scheme–object

C++ Type: array

External Rep: #(element1 element2 element3 ...)

Example:

```
; vector (data type)
; Create and inquire a vector object.
(define myvector (make–vector 10))
;; myvector
(vector? myvector)
;; #t
(vector 'a 'b 'c)
;; #(a b c)
```

version_tag

Scheme Data Type: Model Topology, Model Object

Description: A Version_Tag is a Scheme data type that points to an AcisVersion class object. This data type is used for passing version information to selected scheme extensions. This data type is created by the versiontag Scheme extension.

Derivation: integer: scheme–object

C++ Type: law

External Rep: #[Major=%d Minor=%d Point=%d Tag=%d]

Example:

```
; version_tag (data type)
; Define the version tag to use.
(define acis_70 (versiontag 7 0 0))
;; acis_70
; #[Major=7 Minor=0 Point=0 Tag=70000]
(define version_of_exec (versiontag))
;; version_of_exec
(define another_version (versiontag 8))
;; another_version
```

vertex

Scheme Data Type: Model Topology, Model Object

Description: A vertex is a topological entity representing the end of one or more edges. Vertex refers to a point object in space, and to the edges that it bounds. Other edges are found by following pointers through coedges. vertex objects are saved and restored as part of the model.

Derivation: vertex : entity : scheme–object

C++ Type: VERTEX

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID

Example:
```
; vertex (data type)
; Create and inquire a vertex object.
(define my_block (solid:block
    (position 4 8 –1) (position 35 35 35)))
;; my_block
(define my_vertices (entity:vertices my_block))
;; my_vertices
(vertex? (car (cdr (cdr (cdr my_vertices)))))
;; #t
```

view

Scheme Data Type: Viewing

Description: A view tracks the mapping from the 3D model to a 2D screen, to files, or to printers. view objects are not saved and restored as part of the model. Each view is associated with a single window on the screen, and each graphics window can contain only a single view. Several views can exist at once, but only one of them can be active. A view can only display a single part. However, a single part can be displayed in multiple views.

A view is completely specified with the following data items:

– A target point on the model. This appears at the center of the window. Default #[position 0 0 0].
– The eye position of the viewer. Default #[position 0 0 500]. The vector created from the target point to the eye position is always perpendicular to and coming out of the computer screen.
– A vector for the up direction of the view. Default #[gvector 0 1 0]. This vector in the default model coordinate system points vertically towards the top of the window. The up vector cannot be parallel to the vector from the target point to the eye position.
– A flag that indicates whether to create a perspective or orthographic view.
– The width and height of the view window given in model space coordinates.

The default axis orientation is: *x*–axis points to the left, *y*–axis points up, and *z*–axis points toward the viewer. The default target point is the origin of the coordinate system. The default eye position is along the *z*–axis of the coordinate system.

A vector always exists between the eye position and the target point, and this vector is always perpendicular to and coming out of the computer screen. Therefore, moving just the eye position has the effect of turning or flipping the part to view it from another angle. Moving just the target point has the effect of moving a portion of the part to the center of the view window.

Each view is associated with either a unique window on the screen or a unique file. Views are not saved when a part is saved. Each view is represented externally by the notation #[view n], where *n* is the window handle or file pointer. The number can be used to reference a view, such as through the command view:with–handle.

Derivation: view : scheme–object

C++ Type: view3d

External Rep: #[view %d] or #[deleted view %d]
where the integer is the window handle or file pointer.

Example:
```
; view (data type)
; Create and inquire a view object.
(define front (view:dl))
;; front
(view? front)
;; #t
```

vradius

Scheme Data Type: Blending

Description: A vradius is an object that is used for defining a variable radius function for advanced blending. vradius objects are not saved and restored as part of the model.

Derivation: vradius: scheme–object

C++ Type: var_radius

External Rep: #[%s vradius %x]
where the string is the vradius type name,
and the hexadecimal number is the memory location of the vradius object.

Example:
```
; vradius (data type)
; Create a vradius object (use elliptical radius)
(define my_rad (abl:ell–rad #t 10 20 45))
;; my_rad
```

wcs

Scheme Data Type: Work Coordinate Systems

Description: A wcs (work coordinate system) is an entity that modifies the transform of newly–created entities. For example, if new entities need to be created on a plane tilted with respect to the model coordinate system, a wcs is created and set active. Thereafter, all entities created are created with respect to the wcs coordinate system, not the model coordinate system.

Derivation: wcs : entity : scheme–object

C++ Type: WCS

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; wcs (data type)
; Create and inquire a wcs object.
(define my_wcs (wcs (position 0 0 0) (gvector 0 –1 0)
    (gvector 1 0 0)))
;; my_wcs
(wcs? my_wcs)
;; #t
```

wire

Scheme Data Type: Model Topology, Model Object

Description: A wire is a topological entity that is a collection of edges and vertices. Wires typically represent profiles, construction lines, and center lines of swept shapes. Wires can also represent wire frames that, when surfaced, form shells. wire objects are saved and restored as part of the model.

Derivation: wire : entity : scheme–object

C++ Type: WIRE

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; wire (data type)
; Create and inquire a wire object.
(define my_edge (circular–edge
    (position 0 0 0) 25 180 270))
;; my_edge
(define my_body (wire–body my_edge))
;; my_body
(define my_wires (entity:wires my_body))
;; my_wires
(wire? (car my_wires))
;; #t
```

wire–body

Scheme Data Type: Model Topology, Model Object

Description: A wire–body is a topological entity that is a body consisting of wires (as opposed to lumps). Wire bodies contain wires, loops, coedges, edges, and vertices. wire–body objects are saved and restored as part of the model.

Derivation: wire–body : body : entity : scheme–object

C++ Type: BODY

External Rep: #[entity %d %d]
where the first integer is the entity ID,
and the second integer is the part ID.

Example:

```
; wire–body (data type)
; Create and inquire a wire–body object.
(define my_edge (edge:circular
    (position 0 0 0) 25 180 270))
;; my_edge
(define my_body (wire–body my_edge))
;; my_body
(wire–body? my_body)
;; #t
```