Chapter 13.
Scheme Extensions Fa thru Hz

Topic: Ignore

face:bs

Scheme Extension: Debugging

Action: Returns the B–spline approximation information for a face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: `(face:bs in-face [extra-info=#f])`

Arg Types:

in–face	face
extra–info	boolean

Returns: real | real...

Errors: –1 when there is no B–spline to evaluate.

Description: Returns the number of control points in *u* and *v*.

in–face specifies the face to be queried.

extra–info is an optional argument. If it is set to true (#t), then additional B–spline information is returned. The default value is false (#f).

Limitations: None

Example:

```
; face:bs
; Create topology to demonstrate command.
(define path (edge:spline (list (position 0 0 0)
    (position 10 0 0) (position 10 10 0))))
;; path
(define profile (edge:ellipse
    (position 0 0 0) (gvector 1 0 0)
    (gvector 0 0 1)))
;; profile
(define pipe (sweep:law profile path))
;; pipe
(define face (list-ref (entity:faces pipe) 0))
;; face
; Get the B-spline approximation information.
(face:bs face)
;; (14 20)
```

face:check

Scheme Extension: Debugging

Action: Determines if a face contains invalid loops.

Filename: kern/kern_scm/loop_scm.cxx

APIs: api_check_face_loops

Syntax: (**face:check** face)

Arg Types: face entity

Returns: boolean

Errors: None

Description: This returns text indicating how many of the various kinds of loops there are in the given face and a Boolean flag indicating whether the check was successful or not. Valid loop types include periphery loops, holes, *u*–separation loops, *v*–separation loops, unknown loops, and “Closed face, no loop”.

face specifies a face entity.

Limitations: None

Example:

```
; face:check
; Create a face.
(define face1 (face:law "vec(cos(x), y, x)"
    -20 (law:eval "10*pi") -10 10))
;; face1
(face:check face1)
; 1 periphery loop.
;; #t
```

face:conical?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a conical face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:conical?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: This extension returns #t if the object is a conical face; otherwise, it returns #f.

object specifies the scheme–object that has to be queried for a conical face.

Limitations: None

Example:

```
; face:conical?
; Create a solid cylinder.
(define cyl1
    (solid:cylinder (position 5 0 0)
    (position 25 25 0) 30))
;; cyl1
; Get the faces of the cylinder.
(define face-list (entity:faces cyl1))
;; face-list
; Determine if the first face is a conical face.
(face:conical? (car face-list))
;; #t
(face:conical? (car (cdr face-list)))
;; #f
```

face:cylinder–axis

Scheme Extension: Construction Geometry

Action: Gets the ray along the axis of a cylindrical–face entity.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:cylinder-axis** entity)

Arg Types: entity cylindrical–face

Returns: ray

Errors: None

Description: The returned ray is a gvector and position that specify the central axis of the cylinder face supplied as the entity input. Note that the input argument is cylinder face and not a solid:cylinder.

entity specifies a cylindrical–face.

Limitations: None

Example:

```
; face:cylinder-axis
; Create a solid cylinder.
(define cyl1
    (solid:cylinder (position 0 0 0)
    (position 8 8 8) 32))
;; cyl1
; Find the faces of the cylinder.
(define faces1 (entity:faces cyl1))
;; faces1
; Determine the axis of a cylindrical face.
(face:cylinder-axis (car faces1))
;; #[ray (4 4 4) (0.57735 0.57735 0.57735)]
```

face:cylinder–radius

Scheme Extension: Construction Geometry

Action: Gets the radius of a cylindrical face entity.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:cylinder-radius** entity)

Arg Types: entity cylindrical–face

Returns: real

Errors: None

Description: The returned real specifies the radius of the cylinder face supplied as the entity input. Note that the input argument is cylinder face and not a solid:cylinder.

entity specifies a cylindrical–face.

Limitations: None

Example:

```
; face:cylinder-radius
; Create a cylinder.
(define cyl1
    (solid:cylinder (position 0 0 0)
    (position 8 8 8) 32))
;; cyl1
; Find the faces of the cylinder.
(define faces1 (entity:faces cyl1))
;; faces1
; (#[entity 3 1] #[entity 4 1] #[entity 5 1])
; Find the radius of the cylindrical face.
(face:cylinder-radius (car faces1))
;; 32
```

face:cylindrical?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a cylindrical face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:cylindrical?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: The returned boolean specifies whether the supplied entity input is a cylindrical face. Note that the input argument is cylinder face and not a solid:cylinder.

object specifies the scheme–object that has to be queried for a cylindrical–face.

Limitations: None

Example:

```
; face:cylindrical?
; Create a solid cylinder.
(define cyl1
    (solid:cylinder (position 0 0 0)
    (position 8 8 8) 32))
;; cyl1
; Find the faces of the cylinder.
(define faces1 (entity:faces cyl1))
;; faces1
; Determine whether cyl1 is a cylindrical face.
(face:cylindrical? cyl1)
;; #f
; Determine whether face 2 is a cylindrical face.
(face:cylindrical? (car faces1))
;; #t
; Determine whether face 3 is a cylindrical face.
(face:cylindrical? (car (cdr faces1)))
;; #f
```

face:derivtest

Scheme Extension: Model Geometry

Action: Tests face quality by comparing the procedural derivatives with finite difference derivatives up to the 4th derivatives.

Filename: kern/kern_scm/surf_scm.cxx

APIs: None

Syntax:

```
(face:derivtest face [num-u] [num-v] [start-u]
    [end-u] [start-u] [start-v] [end-v] [file])
```

Arg Types:	face	entity
	num–u	integer
	num–v	integer
	start–u	real
	end–u	real
	start–v	real
	end–v	real
	file	string

Returns: string

Errors: None

Description: This Scheme extension tests the face quality by comparing the procedural derivatives with finite difference derivatives up to the 4th derivatives. Output message can be sent to a optional data file.

face defines the face entity to test derivatives.

num–u defines the position number to test in the surface u direction. The default is 10.

num–v defines the position number to test in the surface v direction. The default is 10.

start–u defines the start surface u parameter. The default is the surface u parameter range.

end–u defines the end surface u parameter.

start–v defines the start surface v parameter. The default is the surface v parameter range.

end–v defines the end surface v parameter.

file defines the output file name. The default is debug_file_ptr.

Limitations: None

Example:

```
; face:derivtest
; Example not available at this time.
```

face:planar?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a planar face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:planar?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: This extension returns #t if the specified object is a planar face.

object specifies the scheme–object that has to be queried for a planar face.

Limitations: None

Example:

```
; face:planar?
; Create a solid block.
(define block1
    (solid:block (position -10 -10 0)
    (position 25 25 25)))
;; block1
; Get a list of the solid block's faces.
(define faces1 (entity:faces block1))
;; faces1
; Determine if one of these faces is
; actually a planar face.
(face:planar? (car (cdr (cdr faces1))))
;; #t
```

face:plane–normal

Scheme Extension: Construction Geometry

Action: Gets the normal of a planar face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:plane-normal** entity)

Arg Types: entity planar–face

Returns: gvector

Errors: None

Description: This extension returns the normal of a planar face.

entity specifies a face entity.

Limitations: None

Example:

```
; face:plane-normal
; Create a solid block.
(define block1
    (solid:block (position 0 0 0)
    (position 40 40 40)))
;; block1
; Get a list of the solid block's faces.
(define faces1 (entity:faces block1))
;; faces1
; Get the normal of one of the planar faces.
(face:plane-normal (car (cdr faces1)))
;; #[gvector 0 0 -1]
; Get the normal of another planar face.
(face:plane-normal (car (cdr (cdr (cdr faces1)))))
;; #[gvector -1 0 0]
```

face:plane–ray

Scheme Extension: Construction Geometry

Action: Gets the plane from a planar face as a ray.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:plane-ray** entity)

Arg Types: entity planar–face

Returns: gvector

Errors: None

Description: This extension represents the specified planar face as a ray.

entity specifies a face entity.

Limitations: None

Example:

```
; face:plane-ray
; Create a solid block.
(define block1
    (solid:block (position 0 0 0)
    (position 40 40 40)))
;; block1
; Get a list of the solid block's faces.
(define faces1 (entity:faces block1))
;; faces1
; Extract a plane from one of the faces and
; represent the face as a ray.
(face:plane-ray (car (cdr faces1)))
;; #[ray (20 20 0) (0 0 -1)]
; Do the same with a second face.
(face:plane-ray (car (cdr (cdr (cdr faces1)))))
;; #[ray (0 20 20) (-1 0 0)]
```

face:scar?

Scheme Extension: Debugging

Action: Checks the input body or face for scars and returns list (or unspecified if no scars exist).

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: `(face:scar? face | body)`

Arg Types:

face	face | face ...
body	body | body ...

Returns: (edge | edge ...) | unspecified

Errors: None

Description: Refer to Action.

face specifies a face or a list of faces.

body specifies a body or a list of bodies.

Limitations: None

Example:

```
; face:scar?
; Create four types of face/edge geometry to
; demonstrate command.
(define block1 (solid:block -40 -5 -15 -25 5 15))
;; block1
(define edge (edge:linear (position -30 0 0)
    (position -30 0 10)))
;; edge
(define body1 (hh:combine (list block1 edge)))
;; body1
(face:scar? block1)
;; ()
; Create a planar disk.
(define pdisk (face:planar-disk
    (position 0 0 0) (gvector 0 0 10) 10))
;; pdisk
(define disk-edge (edge:linear
    (position -10 0 0) (position 10 0 0)))
;; disk-edge
(define body2 (hh:combine (list pdisk disk-edge)))
;; body2
(face:scar? body2)
;; ()
(define block2 (solid:block 20 10 0 30 20 40))
;; block2
(define block2-edge (edge:linear
    (position 27 10 0) (position 22 15 20)))
;; block2-edge
(define body3 (hh:combine (list block2 block2-edge)))
;; body3
(define cylinder (solid:cylinder
    (position -5 0 -14) (position -5 0 -34) 5))
;; cylinder
(define cyl-edge (edge:linear
    (position -3 5 -14) (position -3 5 -35)))
;; cyl-edge
(define body4 (hh:combine (list cylinder cyl-edge)))
;; body4
```

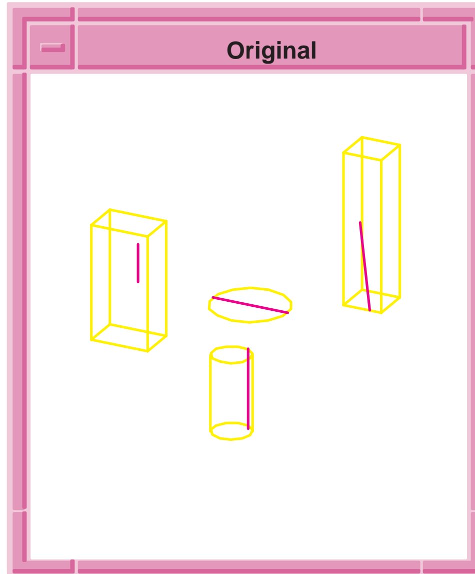

Figure 13-1. face:scar?

face:sphere–center

Scheme Extension: Construction Geometry

Action: Gets the center position of a spherical face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:sphere-center** face)

Arg Types: face spherical–face

Returns: position

Errors: None

Description: This extension returns the position of the center of a spherical face.

face specifies a spherical face entity.

Limitations: None

Example:

```
; face:sphere-center
; Create a solid sphere.
(define sphere1 (solid:sphere (position 0 0 0) 38))
;; sphere1
; Find the faces of the solid sphere.
(define faces1 (entity:faces sphere1))
;; faces1
; Find the center of the spherical face.
(face:sphere-center (car faces1))
;; #[position 0 0 0]
```

face:sphere–radius

Scheme Extension: Construction Geometry

Action: Gets the radius of a spherical face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:sphere-radius** face)

Arg Types: face spherical–face

Returns: real

Errors: None

Description: This extension returns the radius of the spherical face.

face specifies a spherical face entity.

Limitations: None

Example:

```
; face:sphere-radius
; Create a solid sphere.
(define sphere1 (solid:sphere (position 0 0 0) 38))
;; sphere1
; Find the faces of the solid sphere.
(define faces1 (entity:faces sphere1))
;; faces1
; Find the radius of a spherical face.
(face:sphere-radius (car faces1))
;; 38
```

face:spherical?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a spherical face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:spherical?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: This extension returns #t if the specified object is a spherical face.

object specifies the scheme–object that has to be queried for a spherical face.

Limitations: None

Example:
```
; face:spherical?
; Create a solid sphere.
(define sphere1 (solid:sphere (position 0 0 0) 20))
;; sphere1
; Determine if the solid sphere is a
; spherical face.
(face:spherical? sphere1)
;; #f
; Find the faces of the solid sphere.
(define faces1 (entity:faces sphere1))
;; faces1
; Determine if the face is actually a
; spherical face.
(face:spherical? (car faces1))
;; #t
```

face:spline?

Scheme Extension: Model Geometry, Spline Interface

Action: Determines if a Scheme object is a face:spline.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:spline?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: Refer to Action.

object specifies the scheme–object that has to be queried for a face spline.

Limitations: None

Example:

```
; face:spline?
; Define a spline edge 1.
(define e1 (edge:spline (list (position 0 0 0)
    (position 20 -20 0) (position 20 0 0))))
;; e1
; Define linear edge 2.
(define e2 (edge:linear (position 20 0 0)
    (position 20 20 0)))
;; e2
; Define linear edge 3.
(define e3 (edge:linear (position 20 20 0)
    (position 0 20 0)))
;; e3
; Define linear edge 4.
(define e4 (edge:linear (position 0 20 0)
    (position 0 0 0)))
;; e4
; Define a wire body from
; the spline and linear edges.
(define w (wire-body (list e1 e2 e3 e4)))
;; w
; Create a solid by sweeping
; a planar wire along a vector.
(define ws (solid:sweep-wire w (gvector 0 0 20)))
;; ws
; Get the faces of the solid.
(define edges1 (entity:faces ws))
;; edges1
; Determine if one of the faces is a spline face.
(face:spline? (car (cdr edges1)))
;; #f
; Determine if another face is a spline face.
(face:spline? (car (cdr (cdr (cdr edges1)))))
;; #t
```

face:toroidal?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a toroidal face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:toroidal?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: Refer to Action.

object specifies the scheme–object that has to be queried for a toroidal face.

Limitations: None

Example:

```
; face:toroidal?
; Create solid torus 1.
(define torus1
    (solid:torus (position -10 -10 -10) 7 3))
;; torus1
; Get a list of the faces on torus 1.
(define faces1 (entity:faces torus1))
;; faces1
; Determine if the face is a toroidal face.
(face:toroidal? (car faces1))
;; #t
```

face:type

Scheme Extension: Debugging

Action: Returns the type of a face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face:type** face1)

Arg Types: face1 entity

Returns: string

Errors: None

Description: This returns a string that tells what type of face has been produced. Output strings include “Plane”, “Cylinder”, “Cone”, “Sphere”, “Torus”, “Spline”, and “Unknown type”. When the face is a spline, it also returns the subtype for the spline.

face1 specifies a face entity.

Limitations: None

Example:

```
; face:type
; Create a face.
(define face1 (face:law "vec (cos (x), y, x)"
    -20 (law:eval "10*pi") -10 10))
;; face1
(face:type face1)
;; "Spline surface (lawsur-spline)"
```

face:types

Scheme Extension: Debugging

Action: Prints a table of all faces in the current part, including their containing entities and surface types.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: api_get_active_part, api_get_faces

Syntax: (**face:types**)

Arg Types: None

Returns: string

Errors: None

Description: Refer to Action.

Limitations: None

Example:

```
; face:types
; create a solid cylinder
(define cylinder (solid:cylinder (position 0 0 0)
    (position 0 0 30) 10))
;; cylinder
; request a list of all faces in current part
(face:types)
; entity:(entity 1 1)
;           face:(entity 4 1) face-type:Cylinder
;           face:(entity 5 1) face-type:Plane
;           face:(entity 6 1) face-type:Plane
;; #t
```

face?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a face.

Filename: kern/kern_scm/qfac_scm.cxx

APIs: None

Syntax: (**face?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: The extension returns #t if the object is a face; otherwise, it returns #f.

object specifies the scheme–object that has to be queried for a face.

Limitations: None

Example:

```
; face?
; Create a solid block.
(define block1
    (solid:block (position -10 -5 -15)
    (position 10 5 15)))
;; block1
; Get the block's faces.
(define faces1 (entity:faces block1))
;; faces1
(face? block1)
;; #f
; Determine if face 2 is a face.
(face? (car (cdr faces1)))
;; #t
```

filter:and

Scheme Extension: Filtering

Action: Computes an AND of two or more entity–filters.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:and** filt1 ... filtn)

Arg Types:

filt1	entity–filter
filtn	entity–filter

Returns: entity–filter

Errors: None

Description: Multiple filters can be combined using the Boolean and filter to form a single filter that can be applied to a single entity or a list of entities. An entity is selected if all parts of the combined filter return #t.

filt1 is an entity–filter. The ellipsis (...) indicates one or more entity–filters.

Limitations: None

Example:

```
; filter:and
; Create solid block 1.
(define block1
    (solid:block (position 10 0 10)
    (position 20 30 40)))
;; block1
; Create linear edge 2.
(define edge1 (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; edge1
; Create circular edge 3.
(define edge2 (edge:circular (position 0 0 0) 20))
;; edge2
; Change the color of the existing entities to red.
(entity:set-color (part:entities) 1)
;; ()
; Create solid sphere 4.
(define sphere1 (solid:sphere
    (position 20 30 40) 30))
;; sphere1
; Create solid sphere 5.
(define cyl1 (solid:cylinder
    (position 40 0 0) (position 5 5 5) 8))
;; cyl1
; Create linear edge 6.
(define edge3 (edge:linear (position 0 50 0)
    (position 50 50 0)))
;; edge3
; Create spline edge 7.
(define edge4 (edge:spline (list
    (position 20 20 20) (position 10 20 30)
    (position 50 40 10))))
;; edge4
; Define a filter for red curves.
(define red-curves (filter:and (filter:color 1)
    (filter:type "edge:circular?")))
;; red-curves
; List the red curve entities.
(filter:apply red-curves (part:entities))
;; (#[entity 4 1])
; The following accomplishes the same thing.
(part:entities red-curves)
;; (#[entity 4 1])
```

filter:apply

Scheme Extension: Filtering

Action: Applies a filter to an entity or list of entities.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:apply** filter entity-or-list)

Arg Types:

filter	entity–filter
entity–or–list	entity | (entity ...)

Returns: (entity ...)

Errors: None

Description: Once a filter is created, the filter can be applied to obtain the particular results. For example, if numerous entities are components of a part of various colors, applying a color filter to the list of entities returns the list of entities that match the filter's color. When applying the filter to an entity that does not meet the requirements for the filter, this extension returns the empty list.

filter specifies an entity–filter.

entity–or–list specifies an entity or an entity list.

Limitations: None

Example:

```
; filter:apply
; Create solid block 1.
(define block1
    (solid:block (position 10 0 10)
    (position 20 30 40)))
;; block1
; Create linear edge 2.
(define edge1 (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; edge1
; Create circular edge 3.
(define edge2 (edge:circular (position 0 0 0) 20))
;; edge2
; Change the color of the entities so far to red.
(entity:set-color (part:entities) 1)
;; ()
```

```
; Create solid sphere 4.
(define sphere1 (solid:sphere
    (position 20 30 40) 30))
;; sphere1
; Create solid sphere 5.
(define cyl1 (solid:cylinder
    (position 40 0 0) (position 5 5 5) 8))
;; cyl1
; Create linear edge 6.
(define edge3 (edge:linear (position 0 50 0)
    (position 50 50 0)))
;; edge3
; Create spline edge 7.
(define edge4 (edge:spline (list
    (position 20 20 20) (position 10 20 30)
    (position 50 40 10))))
;; edge4
; Apply a green filter and obtain the entities.
(filter:apply (filter:color 2) (part:entities))
;; (#[entity 5 1] #[entity 6 1]
;; #[entity 7 1] #[entity 8 1])
; Apply a solid, red filter and obtain the entities.
(filter:apply (filter:and (filter:type "solid?")
    (filter:color 1)) (part:entities))
;; (#[entity 2 1])
; Apply a solid, green filter and
; obtain the entities.
(part:entities (filter:type "solid?"))
;; (#[entity 2 1] #[entity 5 1] #[entity 6 1])
(filter:apply (filter:type "solid?") edge1)
;; ()
```

filter:not

Scheme Extension: Filtering

Action: Computes the NOT of an input entity–filter.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:not** filter)

Arg Types: filter entity–filter

Returns: entity–filter

Errors: None

Description: Refer to Action.

filter specifies an entity–filter.

Limitations: None

Example:

```
; filter:not
; Create solid block 1.
(define block1
    (solid:block (position 10 0 10)
    (position 20 30 40)))
;; block1
; Create linear edge 2.
(define edge1 (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; edge1
; Create circular edge 3.
(define edge2 (edge:circular (position 0 0 0) 20))
;; edge2
; Change the color of the entities so far to red.
(entity:set-color (part:entities) 1)
;; ()
; Create solid sphere 4.
(define sphere1 (solid:sphere
    (position 20 30 40) 30))
;; sphere1
; Create solid sphere 5.
(define cyl1 (solid:cylinder
    (position 40 0 0) (position 5 5 5) 8))
;; cyl1
; Create linear edge 6.
(define edge3 (edge:linear (position 0 50 0)
    (position 50 50 0)))
;; edge3
; Create spline edge 7.
(define edge4 (edge:spline (list
    (position 20 20 20) (position 10 20 30)
    (position 50 40 10))))
;; edge4
; Apply a green filter and obtain the entities.
(filter:apply (filter:color 2) (part:entities))
;; (#[entity 5 1] #[entity 6 1] #[entity 7 1]
```

```
;; #[entity 8 1])
; Define a yes-red filter.
(define yes-red (filter:color 1))
;; yes-red
(part:entities yes-red)
;; (#[entity 1 1] #[entity 2 1]
;; #[entity 3 1] #[entity 4 1])
; Define a not-red filter.
(define not-red (filter:not (filter:color 1)))
;; not-red
; Apply a not-red filter and obtain the entities.
(part:entities not-red)
;; (#[entity 5 1] #[entity 6 1] #[entity 7 1]
;; #[entity 8 1])
```

filter:or

Scheme Extension: Filtering

Action: Computes the OR of two or more entity–filters.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:or** filt1 ... filtn)

Arg Types:

filt1	entity–filter
filtn	entity–filter

Returns: entity–filter

Errors: None

Description: Multiple filters can be combined using the Boolean or filter to form a single filter that can be applied to a single entity or a list of entities. An entity will be selected if at least one part of the combined filter returns #t.

filt1 is an entity–filter. The ellipsis (...) indicates one or more entity–filters.

Limitations: None

Example:

```
; filter:or
; Create solid block 1.
(define block1
    (solid:block (position 10 0 10)
    (position 20 30 40)))
;; block1
; Create linear edge 2.
(define edge1 (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; edge1
; Create circular edge 3.
(define edge2 (edge:circular (position 0 0 0) 20))
;; edge2
; Change the color of the entities so far to red.
(entity:set-color (part:entities) 1)
;; ()
; Create solid sphere 4.
(define sphere1 (solid:sphere
    (position 20 30 40) 30))
;; sphere1
; Create solid sphere 5.
(define cyl1 (solid:cylinder
    (position 40 0 0) (position 5 5 5) 8))
;; cyl1
; Create linear edge 6.
(define edge3 (edge:linear (position 0 50 0)
    (position 50 50 0)))
;; edge3
; Create spline edge 7.
(define edge4 (edge:spline (list
    (position 20 20 20) (position 10 20 30)
    (position 50 40 10))))
;; edge4
; Define the green-or-solid filter.
(define green-or-solid (filter:or (filter:color 2)
    (filter:type ”solid?”)))
;; green-or-solid
; Apply a green-or-solid filter and
; obtain the entities.
(part:entities green-or-solid)
;; (#[entity 2 1] #[entity 5 1] #[entity 6 1]
;; #[entity 7 1] #[entity 8 1])
```

filter:type

Scheme Extension: Filtering

Action: Creates a filter entity that selects for the type of an entity.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:type** type-name)

Arg Types: type–name string

Returns: entity–filter

Errors: None

Description: This extension creates the specified type–name as a filter, which specifies the type of entity to be used in another filter operation.

If a new type filter is created, it replaces the previously–defined type.

Refer to filter:color for creating filters based on color, and filter:types to display the list of available filter types.

type–name specifies an entity–filter to be created. The possible string values for the type–name are:

“edge:curve?”, “edge:linear?”, “edge:circular?”, “edge:elliptical?”, “edge:spline?”, “edge?”, “body?”, “solid?”, “wire–body?”, “mixed–body?”, “wire?”, “face?”, “face:planar?”, “face:spherical?”, “face:cylindrical?”, “face:conical?”, “face:toroidal?”, “face:spline?”, “wcs?”, “text?”, “vertex?”, or “point?”.

Limitations: None

Example:

```
; filter:type
; Create a solid block.
(define part1
    (solid:block (position 10 0 10)
    (position 20 30 40)))
;; part1
; Create linear edge.
(define part2 (edge:linear (position 0 0 0)
    (position 10 10 10)))
;; part2
; Create circular edge.
(define part3 (edge:circular (position 0 0 0) 20))
```

```
;; part3
; Change the color of the existing entities to red.
(entity:set-color (part:entities) 1)
;; ()
; Create solid sphere.
(define part4 (solid:sphere
    (position 20 30 40) 30))
;; part4
; Create solid cylinder.
(define part5 (solid:cylinder
    (position 40 0 0) (position 5 5 5) 8))
;; part5
; Create another linear edge.
(define part6 (edge:linear (position 0 50 0)
    (position 50 50 0)))
;; part6
; Create a spline edge.
(define part7 (edge:spline (list
    (position 20 20 20) (position 10 20 30)
    (position 50 40 10))))
;; part7
; Get a list of available filter types.
(filter:types)
;; ("point?" "vertex?" "text?" "wcs?" "face:spline?"
;; "face:toroidal?" "face:conical?"
;; "face:cylindrical?" "face:spherical?"
;; "face:planar?" "face?" "wire?" "mixed-body?"
;; "wire-body?" "solid?" "body?" "edge?"
;; "edge:spline?" "edge:elliptical?"
;; "edge:circular?" "edge:linear?" "edge:curve?")
; Apply a solid filter and get entities.
(part:entities (filter:type "solid?"))
;; (#[entity 2 1] #[entity 5 1] #[entity 6 1])
; Apply edge:spline filter and get entities.
(part:entities (filter:type "edge:spline?"))
;; (#[entity 8 1])
```

filter:types

Scheme Extension: Filtering

Action: Gets a list of available filter types.

Filename: kern/kern_scm/filt_scm.cxx

APIs: None

Syntax: (**filter:types**)

Arg Types: None

Returns: (string ...)

Errors: None

Description: This extension returns all the valid filter types as a list of strings.

Limitations: None

Example:

```
; filter:types
; Get a list of available filter types.
(filter:types)
;; ("point?" "vertex?" "text?" "wcs?" "face:spline?"
;; "face:toroidal?" "face:conical?"
;; "face:cylindrical?" "face:spherical?"
;; "face:planar?" "face?" "wire?" "mixed-body?"
;; "wire-body?" "solid?" "body?" "edge?"
;; "edge:spline?" "edge:elliptical?"
;; "edge:circular?" "edge:linear?" "edge:curve?")
```

find:angle

Scheme Extension: Physical Properties

Action: Returns the angle between edges. Returns a list of angles if a non–branched wire–body is submitted.

Filename: kern/kern_scm/find_scm.cxx

APIs: api_get_edges

Syntax: (**find:angle** input1 [input2] [logical])

Arg Types:

input1	vertex | edge | wire–body
input2	edge
logical	real

Returns: real | (real ...)

Errors: None

Description: Refer to Action.

input1 specifies a vertex, edge or a wire–body. A vertex as input1 computes the angles between the two edges around the vertex. If input1 is a closed edge, the angle between the start and end is returned. If input1 is a non–branched, wire–body, a list of angles between each of the edges of the wire–body is returned.

input2 specifies an edge. input2 must be supplied if input1 is an open edge. The angle between these two edges is returned.

A logical of false (#f) returns the results in radians, the default is degrees.

Limitations: Success is not guaranteed for branched wire–bodies, edges that do not share a vertex, and vertices with more than two edges.

Example:

```
; find:angle
; Create an entity
(define p1 (wire-body:polygon
    (position 0 0 0) (gvector 0 1 0)
    (gvector 0 0 1) 5))
;; p1
(define p2 (wire-body:polygon
    (position 0 2 0) (gvector 0 -1 0)
    (gvector 0 0 1) 5))
;; p2
(define unite (bool:unite p1 p2))
;; unite
(zoom-all)
;; #[view 25363466]
(define v (list-ref (entity:vertices p1)3))
;; v
(entity:set-color v 1)
;; ()
(find:angle v)
;; 108.0
```

find:bump

Scheme Extension: Physical Properties

Action: Finds the bump associated with the given face or loop.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_pattern_find_bump

Syntax:

```
(find:bump seed [return-type [no-cross-list
    [show-loop=#f]]])
```

Arg Types:

seed	entity
return–type	string
no–cross–list	entity | (entity ...)
show–loop	boolean

Returns: entity ...

Errors: None

Description: Finds the bump associated with the face or loop specified by seed, and highlights the face of the bump in red.

seed specifies the entity to be searched.

return–type is an optional argument that could be used to have the function return a list of entities in the bump. The options for return–type are ”faces”, ”loops”, and ”all”. ”faces” returns a list of all faces in the bump. “loops” returns a list of all loops in the bump (e.g., those not owned by faces on the bump). ”all” returns a list consisting of both the above. No list is returned unless this string is present.

no–cross–list allows for finer definition or limitation in the search.

show–loop set to true (#t), highlights any limiting loops on the bump in yellow.

Limitations: None

Example:

```
; find:bump
; create a bump
(define blank (solid:block (position 0 0 0)
    (position 10 10 -1)))
;; blank
(define tool (solid:block (position 1 1 0)
    (position 2 2 1)))
;; tool
(define unite (solid:unite blank tool))
;; unite
; pick out one face on the bump
(define bump_face (car (entity:faces blank)))
;; bump_face
; pass in an empty string and list so that
; we highlight the default faces and loops
; belonging to the bump, but return no list
(find:bump bump_face "" (list ) #t)
;; ()
; loop:(entity 14 1)
; face:(entity 3 1)
; face:(entity 7 1)
; face:(entity 4 1)
; face:(entity 5 1)
; face:(entity 6 1)
```

find:pattern–index

Scheme Extension: Patterns

Action: Finds the pattern index associated with a given entity.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**find:pattern-index** entity)

Arg Types: entity entity

Returns: integer

Errors: An invalid entity was specified.

Description: Finds the zero–based pattern index associated with the entity specified by entity.

entity specifies the entity to be searched.

Limitations: None

Example:

```
; find:pattern-index
; make a prism
(define height 1)
;; height
(define maj_rad 1)
;; maj_rad
(define min_rad 0.5)
;; min_rad
(define num-sides 3)
;; num-sides
(define prism (solid:prism height maj_rad min_rad
    num-sides))
;; prism
; position the prism
(define origin (position 1 2 3))
;; origin
(define transform (entity:transform prism
    (transform:axes origin
    (gvector 1 0 0) (gvector 0 1 0))))
;; transform
; make a pattern
(define center origin)
;; center
(define normal (gvector 0 0 1))
```

```
;; normal
(define num-radial 4)
;; num-radial
(define num-angular 5)
;; num-angular
(define spacing 3)
;; spacing
(define pat (pattern:radial center normal
    num-radial num-angular spacing))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
; find the pattern index of a specific lump
(define lump (list-ref (entity:lumps body) 8))
;; lump
(define index (find:pattern-index lump))
;; index
; check the index
(law:equal-test index 8)
;; #t
```

graph:add–edge

Scheme Extension: Graph Theory

Action: Adds an edge to a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:add-edge** output-graph vertex1 vertex2)

Arg Types:

output–graph	graph
vertex1	string | entity
vertex2	string | entity

Returns: graph

Errors: None

Description: This extension adds an edge to an existing graph between two existing vertices.

output–graph specifies a graph. The output–graph is updated to show the new connection between vertices.

The vertex1 and vertex2 elements are required to be part of the output–graph. If the output–graph was created using face entities as the vertices, the vertex1 and vertex2 can be either the face entities or their designation as part of the graph.

Limitations: None

Example:

```
; graph:add-edge
; Create a simple example
(define g1 (graph "me-you us-them"))
;; g1
; Add a new edge between two existing vertices
(define g2 (graph:add-edge g1 "me" "them"))
;; g2
```

graph:add–vertex

Scheme Extension: Graph Theory

Action: Adds a vertex to a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:add-vertex** in-graph in-name)

Arg Types:

in–graph	graph
in–name	string

Returns: graph

Errors: None

Description: This adds the in–name string as a vertex in in–graph.

in–graph specifies a graph.

in–name is a string specifying the vertex that has to be added to the in–graph.

Limitations: None

Example:

```
; graph:add-vertex
; Create a simple example
(define g1 (graph "me-you us-them"))
;; g1
; Add a vertex.
(define g2 (graph:add-vertex g1 "NEW_ONE"))
;; g2
; CAREFUL: The order of the graph output may
; not be the same each time.
; Create an example using entities.
(define b1 (solid:block (position -5 -10 -20)
    (position 5 10 15)))
;; b1
(define faces1 (entity:faces b1))
;; faces1
; Turn the block faces into vertices of the graph.
(define g3 (graph faces1))
;; g3
; Add a vertex.
(define g4 (graph:add-vertex g3 "NEW_ONE"))
;; g4
```

graph:adjacent

Scheme Extension: Graph Theory

Action: Returns whether or not two vertices in a graph are connected with an edge.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:adjacent** in-graph vertex1 vertex2)

Arg Types:

in–graph	graph
vertex1	string | entity
vertex2	string | entity

Returns: boolean

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

The vertex1 and vertex2 elements are required to be part of the in–graph. They could be either entities or their designation as part of the in–graph.

Limitations: None

Example:
```
; graph:adjacent
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:adjacent g1 "we" "FIDO")
;; #f
(graph:adjacent g1 "we" "they")
;; #t
```

graph:branch

Scheme Extension: Graph Theory

Action: Returns a subgraph of the given input graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax:
```
(graph:branch in-graph in-trunk
    which-branch [keep-trunk=#f])
```

Arg Types:

in–graph	graph
in–trunk	graph
which–branch	integer
keep–trunk	boolean

Returns: graph

Errors: None

Description: This command returns a subgraph of the given in–graph that is made up of all the branches that are connected to a given vertex in the ordered in–trunk graph.

in–graph specifies a graph.

in–trunk specifies a graph showing all the connections to a given vertex.

which–branch is an integer signifying the vertex to be used.

The keep–trunk is an option to keep (#t) or not keep (#f) the vertex from the trunk.

Limitations: The in–trunk must be a linear ordered subgraph of the in–graph. The which–branch must be a nonnegative integer less than the max order of the trunk.

Example:

```
; graph:branch
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(define g2 (graph "b-c"))
;; g2
(graph:order-from g2 "b")
;; 1
(graph:branch g1 g2 0)
;; #[graph "a"]
(graph:branch g1 g2 0 #t)
;; #[graph "a-b"]
(graph:branch g1 g2 1)
;; #[graph "f-g f-h d e"]
(graph:branch g1 g2 1 #t)
;; #[graph "c-d c-e c-f f-g f-h"]
```

graph:component

Scheme Extension: Graph Theory

Action: Creates a new graph from all of the component elements of a given graph specified by one of the component elements.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:component** in-graph in-which)

Arg Types:

in–graph	graph
in–which	integer | string | entity

Returns: graph

Errors: None

Description: This extension is useful if the given in–graph has multiple components. It creates a new graph from just the elements of a single component.

in–graph specifies a graph.

in–which specifies a component. The component is selected by providing the integer of the component (numbering starts at 0), a string which is the name of an element of the component, or an entity that is associated with an element of the component.

Limitations: None

Example:

```
; graph:component
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:components g1)
;; 3
(define g2 (graph:component g1 "me"))
;; g2
(define g3 (graph:component g1 "FIDO"))
;; g3
(define g4 (graph:component g1 1))
;; g4
```

graph:components

Scheme Extension: Graph Theory

Action: Returns the number of independent components that are in a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:components** in-graph)

Arg Types: in–graph graph

Returns: integer

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:components
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:components g1)
;; 3
(define g2 (graph:component g1 "me"))
;; g2
(define g3 (graph:component g1 "FIDO"))
;; g3
(define g4 (graph:component g1 1))
;; g4
```

graph:connected?

Scheme Extension: Graph Theory

Action: Determines whether or not the specified graph is connected, or all in one component.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:connected?** in-graph)

Arg Types: in–graph graph

Returns: boolean

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:connected?
; Create a simple example
(define g1 (graph ”me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY”))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:connected? g1)
;; #f
(graph:components g1)
;; 3
(define g2 (graph:component g1 ”me”))
;; g2
(define g3 (graph:component g1 ”FIDO”))
;; g3
(define g4 (graph:component g1 1))
;; g4
(graph:connected? g4)
;; #t
```

graph:copy

Scheme Extension: Graph Theory

Action: Creates a new graph that is a copy of the specified graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:copy** in-graph)

Arg Types: in–graph graph

Returns: graph

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:copy
; Create a simple example
(define g1 (graph ”me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY”))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:component g1 ”FIDO”))
;; g2
(define g3 (graph:copy g2))
;; g3
; CAREFUL: The order may not be the same as the
; original, but graphs are still equivalent.
```

graph:cut–edge?

Scheme Extension: Graph Theory

Action: Determines whether or not the specified edge is a cutting edge.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cut-edge**? in-graph in-edge)

Arg Types:

in–graph	graph
in–edge	string

Returns: boolean

Errors: None

Description: A cutting edge is an edge whose removal creates more components in the graph than are present when the edge is not removed.

in–graph specifies a graph.

in–edge specifies the edge to be queried.

Limitations: None

Example:

```
; graph:cut-edge?
; Create a simple example
(define g1 (graph ”me-you us-them
    we-they them-they we-me us-me”))
;; g1
; them-us they-we”]
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:cut-edges g1))
;; g2
(graph:cut-edge? g1 ”us-them”)
;; #f
(graph:cut-edge? g1 ”me-you”)
;; #t
```

graph:cut–edges

Scheme Extension: Graph Theory

Action: Returns all of the cutting edges of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cut-edges** in-graph)

Arg Types: in–graph graph

Returns: graph

Errors: None

Description: A cutting edge is an edge whose removal creates more components in the graph than are present when the edge is not removed.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:cut-edges
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they we-me us-me"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:cut-edges g1))
;; g2
(graph:cut-edge? g1 "us-them")
;; #f
(graph:cut-edge? g1 "me-you")
;; #t
```

graph:cut–vertex?

Scheme Extension: Graph Theory

Action: Determines whether or not the specified vertex is a cutting vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cut-vertex?** in-graph test-vertex)

Arg Types:

in–graph	graph
test–vertex	string | entity |

Returns: boolean

Errors: None

Description: A cutting vertex is vertex whose removal creates more components in the graph than are present when the vertex is not removed.

in–graph specifies a graph.

test–vertex could be either the designation string in the graph or an entity associated with that graph vertex.

Limitations: None

Example:

```
; graph:cut-vertex?
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:cut-vertices g1))
;; g2
(graph:cut-vertex? g1 "us")
;; #f
(graph:cut-vertex? g1 "SPOT")
;; #t
```

graph:cut–vertices

Scheme Extension: Graph Theory

Action: Returns all of the cutting vertices of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cut-vertices** in-graph)

Arg Types: in–graph graph

Returns: graph

Errors: None

Description: A cutting vertex is vertex whose removal creates more components in the graph than are present when the vertex is not removed.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:cut-vertices
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:cut-vertices g1))
;; g2
```

graph:cycle–vertex?

Scheme Extension: Graph Theory

Action: Determines whether or not a given vertex is a cycle vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cycle-vertex?** in-graph in-vertex)

Arg Types:

in–graph	graph
in–vertex	string | entity

Returns: boolean

Errors: None

Description: A cycle is defined as a connected group of vertices whose individual removal from the graph results in a linear graph and the same number of components. In other words, none of the vertices of the cycle are cut vertices and none have edges to more than one vertex.

in–graph specifies a graph.

in–vertex could be the designated name string within the graph or the model entity associated with the graph vertex.

Limitations: None

Example:

```
; graph:cycle-vertex?
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:cycle? g1)
;; #f
(define g2 (graph:component g1 "FIDO"))
;; g2
(graph:cycle? g2)
;; #f
(define g3 (graph:component g1 "me"))
;; g3
(graph:cycle? g3)
;; #t
(graph:cycle-vertex? g1 "FIDO")
;; #f
(graph:cycle-vertex? g1 "me")
;; #t
(graph:cycle-vertex? g3 "me")
;; #t
```

graph:cycle?

Scheme Extension: Graph Theory

Action: Determines whether or not a graph has a cycle.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:cycle?** in-graph)

Arg Types: in–graph graph

Returns: boolean

Errors: None

Description: A cycle is defined as a connected group of vertices whose individual removal from the graph results in a linear graph and the same number of components. In other words, none of the vertices of the cycle are cut vertices and none have edges to more than one vertex.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:cycle?
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:cycle? g1)
;; #f
(define g2 (graph:component g1 "FIDO"))
;; g2
(graph:cycle? g2)
;; #f
(define g3 (graph:component g1 "me"))
;; g3
(graph:cycle? g3)
;; #t
```

graph:degree?

Scheme Extension: Graph Theory

Action: Returns the number of graph vertices that are connected with graph edges to the specified vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:degree?** in-graph in-vertex)

Arg Types:

in–graph	graph
in–vertex	string | entity

Returns: integer

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

in–vertex could be either the designation name used as part of the graph or the model entity associated with that graph vertex.

Limitations: None

Example:

```
; graph:degree?
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:degree? g1 "me")
;; 2
(graph:degree? g1 "SPOT")
;; 3
(graph:degree? g1 "PETEY")
;; 1
; Create an example using entities.
(define b1 (solid:block
    (position -5 -10 -20) (position 5 10 15)))
;; b1
(define faces1 (entity:faces b1))
;; faces1
; Turn the block faces into vertices of the graph.
(define g3 (graph faces1))
;; g3
(graph:degree? g3 "(Face 0)")
;; 4
(graph:degree? g3 (list-ref faces1 3))
;; 4
```

graph:edge–entities

Scheme Extension: Graph Theory

Action: Returns a list of model entities associated with the graph edges.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:edge-entities** in-graph)

Arg Types: in–graph graph

Returns: (entity...)

Errors: None

Description: The edges of a graph can have entities associated with them. An example of this is the case of a wirebody. In a wirebody, the vertices of the wireframe become vertices in the graph while the edges of the wirebody become edges in the graph.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:edge-entities
; Create an example using entities.
(define e1 (edge:linear (position 10 10 0)
    (position 10 -10 0)))
;; e1
(define e2 (edge:linear (position 10 -10 0)
    (position -10 -10 0)))
;; e2
(define e3 (edge:linear (position -10 -10 0)
    (position -10 10 0)))
;; e3
(define e4 (edge:linear (position -10 10 0)
    (position 10 10 0)))
;; e4
(define g1 (graph (list e1 e2 e3 e4)))
;; g1
(graph:edge-entities g1)
;; (#[entity 5 1] #[entity 4 1] #[entity 3 1]
;; #[entity 2 1])
(graph:vertex-entities g1)
;; (#[entity 6 1] #[entity 7 1] #[entity 8 1]
;; #[entity 9 1] #[entity 10 1] #[entity 11 1]
;; #[entity 12 1] #[entity 13 1])
(define b1 (solid:block (position -5 -10 -20)
    (position 5 10 15)))
;; b1
(define faces1 (entity:faces b1))
;; faces1
; Turn the block faces into vertices of the graph.
(define g2 (graph faces1))
;; g2
; (entity 16 65536)-(entity 19 65536)
(graph:edge-entities g2)
```

```
;; ()
(graph:vertex-entities g2)
;; (#[entity 20 1] #[entity 19 1] #[entity 18 1]
;; #[entity 17 1] #[entity 16 1] #[entity 15 1])
(define g3 (graph:unite g1 g2))
;; g3
(graph:edge-entities g3)
;; (#[entity 2 1] #[entity 3 1] #[entity 4 1]
;; #[entity 5 1])
(graph:vertex-entities g3)
;; (#[entity 13 1] #[entity 12 1] #[entity 11 1]
;; #[entity 10 1] #[entity 9 1] #[entity 8 1]
;; #[entity 7 1] #[entity 6 1] #[entity 15 1]
;; #[entity 16 1] #[entity 17 1] #[entity 18 1]
;; #[entity 19 1] #[entity 20 1])
```

graph:edge–weight

Scheme Extension: Graph Theory

Action: Sets the weight for an edge of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax:
```
(graph:edge-weight in-graph {edge-name weight} |
    {vertex-name1 vertex-name2 weight})
```

Arg Types:

in–graph	graph
edge–name	string
weight	real
vertex–name1	string
vertex–name2	string

Returns: graph

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

edge–name specifies the edge by name.

weight specifies the value to be assigned.

vertex–name1 and vertex–name2 specifies the edge by naming the two bounding vertices.

Limitations: None

Example:

```
; graph:edge-weight
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:edge-weight g1 "a" "b" 3)
;; #[graph "a-b b-c c-d c-e c-f f-g f-h"]
(graph:edge-weight g1 "c-e" 5)
;; #[graph "a-b b-c c-d c-e c-f f-g f-h"]
(graph:total-weight g1)
;; 8
```

graph:entities

Scheme Extension: Graph Theory

Action: Returns a list of model entities associated with the graph vertices and edges.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:entities** in-graph [use-ordering=#f])

Arg Types:

in–graph	graph
use–ordering	boolean

Returns: (entity ...)

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

If use–ordering is true (#t), sorts the result by graph order. The default value is false (#f).

Limitations: None

Example:

```
; graph:entities
; Create an example using entities.
(define e1 (edge:linear (position 10 10 0)
    (position 10 -10 0)))
;; e1
(define e2 (edge:linear (position 10 -10 0)
    (position -10 -10 0)))
;; e2
(define e3 (edge:linear (position -10 -10 0)
    (position -10 10 0)))
;; e3
(define e4 (edge:linear (position -10 10 0)
    (position 10 10 0)))
;; e4
(define g1 (graph (list e1 e2 e3 e4)))
;; g1
(graph:entities g1)
;; (#[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 5 1] #[entity 4 1] #[entity 3 1]
```

```
;; #[entity 2 1])
(graph:edge-entities g1)
;; (#[entity 5 1] #[entity 4 1] #[entity 3 1]
;; #[entity 2 1])
(graph:vertex-entities g1)
;; (#[entity 6 1] #[entity 7 1] #[entity 8 1]
;; #[entity 9 1] #[entity 10 1] #[entity 11 1]
;; #[entity 12 1] #[entity 13 1])
(define b1 (solid:block (position -5 -10 -20)
    (position 5 10 15)))
;; b1
(define faces1 (entity:faces b1))
;; faces1
; Turn the block faces into vertices of the graph.
(define g2 (graph faces1))
;; g2
(graph:entities g2)
;; (#[entity 20 1] #[entity 19 1] #[entity 18 1]
;; #[entity 17 1] #[entity 16 1] #[entity 15 1])
(graph:edge-entities g2)
;; ()
(graph:vertex-entities g2)
;; (#[entity 20 1] #[entity 19 1] #[entity 18 1]
;; #[entity 17 1] #[entity 16 1] #[entity 15 1])
(define g3 (graph:unite g1 g2))
;; g3
(graph:entities g3)
;; (#[entity 13 1] #[entity 12 1] #[entity 11 1]
;; #[entity 10 1] #[entity 9 1] #[entity 8 1]
;; #[entity 7 1] #[entity 6 1] #[entity 15 1]
;; #[entity 16 1] #[entity 17 1] #[entity 18 1]
;; #[entity 19 1] #[entity 20 1] #[entity 2 1]
;; #[entity 3 1] #[entity 4 1] #[entity 5 1])
(graph:edge-entities g3)
;; (#[entity 2 1] #[entity 3 1] #[entity 4 1]
;; #[entity 5 1])
(graph:vertex-entities g3)
;; (#[entity 13 1] #[entity 12 1] #[entity 11 1]
;; #[entity 10 1] #[entity 9 1] #[entity 8 1]
;; #[entity 7 1] #[entity 6 1] #[entity 15 1]
;; #[entity 16 1] #[entity 17 1] #[entity 18 1]
;; #[entity 19 1] #[entity 20 1])
```

graph:get–order

Scheme Extension: Graph Theory

Action: Returns a number representing the distance a given graph vertex is from the 0 node in the given ordered graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:get-order** in-graph in-vertex)

Arg Types:

in–graph	graph
in–vertex	string | entity

Returns: integer

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

in–vertex could be either the designation name used as part of the graph or the model entity associated with that graph vertex.

Limitations: None

Example:

```
; graph:get-order
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:order-from g1 "a")
;; 4
(graph:get-order g1 "a")
;; 0
(graph:get-order g1 "b")
;; 1
(graph:get-order g1 "h")
;; 4
(graph:show-order g1)
;; ("a 0" "b 1" "c 2" "e 3" "d 3" "f 3" "g 4" "h 4")
```

graph:intersect

Scheme Extension: Graph Theory, Booleans

Action: Performs a Boolean intersect operation of two graphs.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:intersect** in-graph1 in-graph2)

Arg Types:

in–graph1	graph
in–graph2	graph

Returns: graph

Errors: None

Description: Given two graphs, returns a new graph that is a Boolean intersection of the two.

in–graph1 and in–graph2 specifies the graphs to be intersected.

Limitations: None

Example:

```
; graph:intersect
; Create some simple graphs.
(define g1 (graph "I-me me-myself myself-mine I-we
    we-us us-them"))
;; g1
(define g2 (graph "he-she it-thing they-those us-we
    them-us"))
;; g2
(define g3 (graph:intersect g1 g2))
;; g3
```

graph:is–subset

Scheme Extension: Graph Theory

Action: Returns TRUE if the small graph is a subset of the large graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:is-subset** small-graph large-graph)

Arg Types:

small–graph	graph
large–graph	graph

Returns: boolean

Errors: None

Description: Refer to Action.

small–graph specifies the subset graph.

large–graph specifies the graph of which small–graph is a subset.

Limitations: None

Example:

```
; graph:is-subset
; Create a graph
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(define g2 (graph "b-c c-e"))
;; g2
(define g3 (graph "h-i i-j"))
;; g3
(graph:is-subset g2 g1)
;; #t
(graph:is-subset g3 g1)
;; #f
```

graph:kind

Scheme Extension: Graph Theory

Action: Returns a graph containing the input graph elements that are of the specified kind number and specified kind status.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:kind** in-graph kind on-off)

Arg Types:

in–graph	graph
kind	integer
on–off	boolean

Returns: graph

Errors: None

Description: A graph can have multiple kinds assigned to it. Each kind can have a status of #t or #f.

in–graph specifies a graph.

kind is an integer specifying the type.

on–off specifies the kind status.

Limitations: None

Example:

```
; graph:kind
; Create some simple graphs.
(define g (graph "a-b b-c c-d c-e"))
;; g
(graph:set-kind g 3 #t "a-b")
;; #[graph "a-b b-c c-d c-e"]
(graph:set-kind g 3 #t "b-c")
;; #[graph "a-b b-c c-d c-e"]
(graph:kind g 0 #f)
;; #[graph "a-b b-c c-d c-e"]
(graph:kind g 3 #f)
;; #[graph "c-d c-e a b"]
(graph:kind g 3 #t)
; *** Error graph:kind: A bad edge was added
; to a graph
;; #f
(graph:kind? g 3 "a-b")
;; #t
(graph:kind? g 2 "a-b")
;; #f
(graph:kind? g 3 "b-c")
;; #t
(graph:kind? g 3 "c-e")
;; #f
```

```
; Create a selective boolean example.
(define blank (solid:block (position 0 0 0)
    (position 25 10 10)))
;; blank
(define b2 (solid:block (position 5 0 0)
    (position 10 5 10)))
;; b2
(define b3 (solid:block (position 15 0 0)
    (position 20 5 10)))
;; b3
(define subtract1 (solid:subtract blank b2))
;; subtract1
(define subtract2 (solid:subtract blank b3))
;; subtract2
(define tool (solid:cylinder
    (position -5 2.5 5) (position 30 2.5 5)1))
;; tool
(define g (bool:select1 blank tool))
;; g
(define p (graph:kind g 0 #t))
;; p
(entity:set-color (graph:entities p) 6)
;; ()
```

graph:kind?

Scheme Extension: Graph Theory

Action: Returns whether or not a graph with a given edge is of the specified kind.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:kind?** in-graph kind item1 [item2])

Arg Types:

in–graph	graph
kind	integer
item1	string	entity
item2	entity

Returns: boolean

Errors: None

Description: A graph can have any number of kind types assigned to edges of the graph. kind is a integer for the type and can take a Boolean value. If not specified, it is assumed to be #f. The assignment of kind and its value is done on a per edge basis.

This Scheme extension provides flexibility for the types of arguments and how they are used.

in–graph specifies a graph.

kind is an integer representing a type that was assigned to a graph edge.

item1 argument can be either a string or an entity. When it is a string, it is tested to see whether it represents the name of an edge in the graph or a vertex in the graph.

item2 argument is only used when item1 is an entity representing a vertex, in which case item2 must also be an entity representing a vertex.

Limitations: None

Example:

```
; graph:kind?
; Create some simple graphs.
(define g (graph "a-b b-c c-d c-e"))
;; g
(graph:set-kind g 3 #t "a-b")
;; #[graph "a-b b-c c-d c-e"]
(graph:set-kind g 3 #t "b-c")
;; #[graph "a-b b-c c-d c-e"]
(graph:kind? g 3 "a-b")
;; #t
(graph:kind? g 2 "a-b")
;; #f
(graph:kind? g 3 "b-c")
;; #t
(graph:kind? g 3 "c-e")
;; #f

; Create an example using entities.
(define e1 (edge:linear (position 10 10 0)
    (position 10 -10 0)))
;; e1
(define e2 (edge:linear (position 10 -10 0)
    (position -10 -10 0)))
;; e2
(define e3 (edge:linear (position -10 -10 0)
    (position -10 10 0)))
;; e3
(define e4 (edge:linear (position -10 10 0)
    (position 10 10 0)))
;; e4
```

```
(define g1 (graph (list e1 e2 e3 e4)))
;; g1
(define ve (graph:vertex-entities g1))
;; ve
(graph:set-kind g1 0 #t
    (list-ref ve 0) (list-ref ve 1))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:set-kind g1 1 #t
    (list-ref ve 2) (list-ref ve 3))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:set-kind g1 2 #t
    (list-ref ve 4) (list-ref ve 5))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:kind? g1 0 (list-ref ve 0) (list-ref ve 1))
;; #t
(graph:kind? g1 1 (list-ref ve 0) (list-ref ve 1))
;; #f
(graph:kind? g1 2 (list-ref ve 0) (list-ref ve 1))
;; #f
(graph:kind? g1 0 (list-ref ve 2) (list-ref ve 3))
;; #f
(graph:kind? g1 1 (list-ref ve 2) (list-ref ve 3))
;; #t
(graph:kind? g1 2 (list-ref ve 2) (list-ref ve 3))
;; #f
(graph:kind? g1 0 (list-ref ve 4) (list-ref ve 5))
;; #f
(graph:kind? g1 1 (list-ref ve 4) (list-ref ve 5))
;; #f
(graph:kind? g1 2 (list-ref ve 4) (list-ref ve 5))
;; #t
```

graph:kinds?

Scheme Extension: Graph Theory

Action: Returns a list of all the kinds on a vertex or edge.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:kinds?** in-graph item1 [item2])

Arg Types:

in–graph	graph
item1	string | entity
item2	entity

Returns: boolean

Errors: None

Description: Given a graph and a vertex or edge, returns a list containing all kinds on that vertex or edge. A graph can have any number of kind types assigned to edges of the graph. kind is an integer for the type and can take a Boolean value. If not specified, it is assumed to be #f. The assignment of kind and its value is done on a per edge basis.

in–graph specifies a graph.

item1 could be either a string or an entity. When it is a string, it is tested to see whether it represents the name of an edge in the graph or a vertex in the graph.

item2 is only used when item1 is an entity representing a vertex, in which case item2 must also be an entity representing a vertex.

Limitations: None

Example:

```
; graph:kinds?
; Create some simple graphs.
(define g (graph "a-b b-c c-d c-e"))
;; g
(graph:set-kind g 3 #t "a-b")
;; #[graph "a-b b-c c-d c-e"]
(graph:set-kind g 3 #t "b-c")
;; #[graph "a-b b-c c-d c-e"]
(graph:kinds? g "a-b")
;; (#f #f #f #t)
(graph:kinds? g "b-c")
;; (#f #f #f #t)
(graph:kinds? g "c-d")
;; ()
```

graph:lightest–path

Scheme Extension: Graph Theory

Action: Returns a graph representing the lightest path between two vertices of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:lightest-path** in-graph in-vertex1 in-vertex2)

Arg Types:

in–graph	graph
in–vertex1	string | entity
in–vertex2	string | entity

Returns: graph

Errors: None

Description: After all edges have a weight assigned, this Scheme extension returns a graph representing the path with the lightest total weight from one given vertex to another.

in–graph specifies a graph.

in–vertex1 could be either a vertex or a string representing the vertex in the graph.

in–vertex2 could be either a vertex or a string representing the vertex in the graph.

Limitations: All edges of the graph require a weight.

Example:

```
; graph:lightest-path
; Create a simple graph.
(define g1 (graph "a-b1 a-b2 b1-c b2-c c-d"))
;; g1
(graph:edge-weight g1 "a" "b1" 3)
;; #[graph "a-b1 a-b2 b1-c b2-c c-d"]
(graph:edge-weight g1 "a-b2" 5)
;; #[graph "a-b1 a-b2 b1-c b2-c c-d"]
(graph:edge-weight g1 "b1-c" 1)
;; #[graph "a-b1 a-b2 b1-c b2-c c-d"]
(graph:edge-weight g1 "b2-c" 1)
;; #[graph "a-b1 a-b2 b1-c b2-c c-d"]
(graph:edge-weight g1 "c-d" 1)
;; #[graph "a-b1 a-b2 b1-c b2-c c-d"]
(graph:lightest-path g1 "a" "d")
;; #[graph "a-b1 b1-c c-d"]
```

graph:linear?

Scheme Extension: Graph Theory

Action: Determines whether or not a graph is linear.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: `(graph:linear? in-graph)`

Arg Types: in–graph graph

Returns: boolean

Errors: None

Description: This extension returns #t if the graph is linear.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:linear?
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:linear? g1)
;; #f
(define g2 (graph "me-you you-us us-them
    them-they"))
;; g2
(graph:linear? g2)
;; #t
```

graph:negate

Scheme Extension: Graph Theory

Action: Negates an ordered graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: `(graph:negate in-graph)`

Arg Types: in–graph graph

Returns: graph

Errors: None

Description: For noncyclic ordered graphs, the highest numbered vertices are assigned 0 and new numbering for the vertices commences from there. For cyclic ordered graphs, the 0 vertex remains the same, but sequence or direction around the cycle changes.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:negate
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:order-from g1 "a")
;; 4
(graph:show-order g1)
;; ("a 0" "b 1" "c 2" "e 3" "d 3" "f 3" "g 4" "h 4")
(define g2 (graph:negate g1))
;; g2
(graph:show-order g2)
;; ("a 4" "b 3" "c 2" "e 1" "d 1" "f 1" "g 0" "h 0")
; Create a simple cyclic example
(define g3 (graph "me-you you-us us-them
    them-they me-they"))
;; g3
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:cycle? g3)
;; #t
(graph:order-cyclic g3 "me" "you")
;; 4
(graph:show-order g3)
;; ("me 0" "you 4" "us 3" "them 2" "they 1")
(define g4 (graph:negate g3))
;; g4
(graph:show-order g4)
;; ("me 0" "you 1" "us 2" "them 3" "they 4")
```

graph:order–cyclic

Scheme Extension: Graph Theory

Action: Assigns a sequence order to the vertices of a cyclic graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:order-cyclic** in-graph in-first in-last)

Arg Types:

in–graph	graph
in–first	string | entity
in–last	string | entity

Returns: integer

Errors: None

Description: A cycle is defined as a connected group of vertices whose individual removal from the graph results in a linear graph and the same number of components. In other words, none of the vertices of the cycle are cut vertices and none have edges to more than one vertex. The extension returns the number of vertices in the graph.

in–graph specifies a graph.

in–first could be either a vertex or a string representing the vertex in the graph.

in–last could be either a vertex or a string representing the vertex in the graph.

Limitations: None

Example:

```
; graph:order-cyclic
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:component g1 "me"))
;; g2
(graph:cycle? g2)
;; #t
(graph:order-cyclic g2 "me" "them")
;; 4
(graph:show-order g2)
;; ("they 4" "them 3" "us 2" "you 1" "me 0")
```

graph:order–from

Scheme Extension: Graph Theory

Action: Sets the order of a graph starting at 0 for the specified vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:order-from** in-graph in-vertex)

Arg Types:

in–graph	graph
in–vertex	string | entity

Returns: integer

Errors: None

Description: When ordering the graph starting at 0 for the specified in–vertex, each subsequent vertex receives a number based on how far away it is (e.g., how many edges) from the starting vertex. The integer returned is the maximum number of "hops" that one or more vertices are from the starting vertex.

in–graph specifies a graph.

in–vertex could be either the designation string for a vertex of the graph or a model entity associated with the graph vertex.

Limitations: None

Example:

```
; graph:order-from
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:order-from g1 "a")
;; 4
(graph:show-order g1)
;; ("a 0" "b 1" "c 2" "e 3" "d 3" "f 3" "g 4" "h 4")
```

graph:order–with

Scheme Extension: Graph Theory

Action: Sets the order of one graph onto another and rescales the ordering to remove gaps.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:order-with** in-graph1 in-graph2)

Arg Types:

in–graph1	graph
in–graph2	graph

Returns: integer

Errors: None

Description: This extension orders the in–graph1 with respect to in–graph2. The integer returned is the maximum order number.

in–graph1 and in–graph2 specifies the graph.

Limitations: None

Example:

```
; graph:order-with
; Create a simple example
(define g1 (graph "a-b b-c c-d d-e"))
;; g1
(graph:order-from g1 "a")
;; 4
(graph:show-order g1)
;; ("a 0" "b 1" "c 2" "d 3" "e 4")
(graph:negate g1)
;; #[graph "a-b b-c c-d d-e"]
(graph:show-order g1)
;; ("a 4" "b 3" "c 2" "d 1" "e 0")
(define s1 (graph "a c e"))
;; s1
(graph:order-with s1 g1)
;; 2
(graph:show-order s1)
;; ("a 2" "c 1" "e 0")
```

graph:set–kind

Scheme Extension: Graph Theory

Action: Specifies the kind type and its on/off value for an edge of the given graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:set-kind** in-graph kind on-off item1 [item2])

Arg Types:

in–graph	graph
kind	integer
on–off	boolean
item1	string | entity
item2	entity

Returns: graph

Errors: None

Description: A graph can have any number of kind types assigned to edges of the graph. kind is a integer for the type and can take a Boolean on–off value. If not specified, it is assumed to be #f. The assignment of kind and its on–off value is done on a per edge basis.

in–graph specifies a graph.

kind is an integer representing a type that was assigned to an element of the graph, either a vertex or edge.

on–off argument is a boolean used to establish whether that kind number is on or off.

item1 argument could be either a string or an entity. When it is a string, it is tested to see whether it represents the name of an edge in the graph or a vertex in the graph.

item2 is only used when item1 is an entity representing a vertex, in which case item2 must also be an entity representing a vertex.

Limitations: None

Example:

```
; graph:set-kind
; Create a simple graph.
(define g (graph "a-b b-c c-d c-e"))
;; g
(graph:set-kind g 3 #t "a-b")
;; #[graph "a-b b-c c-d c-e"]
(graph:set-kind g 3 #t "b-c")
;; #[graph "a-b b-c c-d c-e"]
(graph:kind? g 3 "a-b")
;; #t
(graph:kind? g 2 "a-b")
;; #f
(graph:kind? g 3 "b-c")
;; #t
(graph:kind? g 3 "c-e")
;; #f
```

```
; Create an example using entities.
(define e1 (edge:linear (position 10 10 0)
    (position 10 -10 0)))
;; e1
(define e2 (edge:linear (position 10 -10 0)
    (position -10 -10 0)))
;; e2
(define e3 (edge:linear (position -10 -10 0)
    (position -10 10 0)))
;; e3
(define e4 (edge:linear (position -10 10 0)
    (position 10 10 0)))
;; e4
(define g1 (graph (list e1 e2 e3 e4)))
;; g1
(define ve (graph:vertex-entities g1))
;; ve
(graph:set-kind g1 0 #t
    (list-ref ve 0) (list-ref ve 1))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:set-kind g1 1 #t
    (list-ref ve 2) (list-ref ve 3))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:set-kind g1 2 #t
    (list-ref ve 4) (list-ref ve 5))
;; #[graph "(entity 10 65536)-(entity 11 65536)
;; (entity 12 65536)-(entity 13 65536)
;; (entity 6 65536)-(entity 7 65536)
;; (entity 8 65536)-(entity 9 65536)"]
(graph:kind? g1 0 (list-ref ve 0) (list-ref ve 1))
;; #t
(graph:kind? g1 1 (list-ref ve 0) (list-ref ve 1))
;; #f
(graph:kind? g1 2 (list-ref ve 0) (list-ref ve 1))
;; #f
(graph:kind? g1 0 (list-ref ve 2) (list-ref ve 3))
;; #f
(graph:kind? g1 1 (list-ref ve 2) (list-ref ve 3))
;; #t
```

```
(graph:kind? g1 2 (list-ref ve 2) (list-ref ve 3))
;; #f
(graph:kind? g1 0 (list-ref ve 4) (list-ref ve 5))
;; #f
(graph:kind? g1 1 (list-ref ve 4) (list-ref ve 5))
;; #f
(graph:kind? g1 2 (list-ref ve 4) (list-ref ve 5))
;; #t
```

graph:shortest–cycle

Scheme Extension: Graph Theory

Action: Returns the shortest cycle graph that includes the specified graph vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:shortest-cycle** in-graph in-vertex)

Arg Types:

in–graph	graph
in–vertex	string | entity

Returns: graph

Errors: None

Description: This extension can be used to trim away branches off of a cyclic graph.

in–graph specifies a graph.

in–vertex could be either a designation string of the graph or a model entity associated with a graph vertex.

Limitations: None

Example:

```
; graph:shortest-cycle
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:shortest-cycle g1 "me"))
;; g2
(define g3 (graph:shortest-cycle g1 "FIDO"))
;; g3
```

graph:shortest–path

Scheme Extension: Graph Theory

Action: Returns the shortest path graph that includes the two specified graph vertices.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:shortest-path** in-graph in-vertex1 in-vertex2)

Arg Types:

in–graph	graph
in–vertex1	string | entity
in–vertex2	string | entity

Returns: graph

Errors: None

Description: This extension can be used to trim away branches off of a cyclic graph.

in–graph specifies a graph.

in–vertex1 could be either a designation string of the graph or a model entity associated with a graph vertex.

in–vertex2 could be either a designation string of the graph or a model entity associated with a graph vertex.

Limitations: None

Example:

```
; graph:shortest-path
; Create a simple example
(define g1 (graph "me-you you-us us-them
    them-they me-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(define g2 (graph:shortest-path g1 "me" "us"))
;; g2
(define g3 (graph:shortest-path g1 "me" "FIDO"))
;; g3
(define g4 (graph:shortest-path g1 "PETEY" "FIDO"))
;; g4
```

graph:show–order

Scheme Extension: Graph Theory

Action: Creates a list of a vertices in a graph and their respective distance from the starting vertex.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:show-order** in-graph)

Arg Types: in–graph graph

Returns: text

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:show-order
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:order-from g1 "a")
;; 4
(graph:show-order g1)
;; ("a 0" "b 1" "c 2" "e 3" "d 3" "f 3" "g 4" "h 4")
```

graph:split–branches

Scheme Extension: Graph Theory

Action: Decomposes a graph into components that do not have branches.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:split-branches** in-graph)

Arg Types: in–graph graph

Returns: graph

Errors: None

Description: This command breaks the branches of a graph into components. It splits the graph into set of subgraphs that are either linear or cyclic with no branches. No edge will belong to more than one subgraph. The union of the subgraphs is the original graph.

in–graph specifies a graph.

Limitations: None

Example:
```
; graph:split-branches;
; Create a simple example
(define block1 (solid:block (position -10 -5 0)
    (position 5 10 15)))
;; block1
(define e (entity:edges block1))
;; e
(define v (entity:vertices block1))
;; v
(define g (graph e))
;; g
(define b-list (graph:split-branches g))
;; b-list
(define g0 (list-ref b-list 0))
;; g0
(define g1 (list-ref b-list 1))
;; g1
(define g2 (list-ref b-list 2))
;; g2
(define g3 (list-ref b-list 3))
;; g3
(define g4 (list-ref b-list 4))
;; g4
(define g5 (list-ref b-list 5))
;; g5
```

graph:subset

Scheme Extension: Graph Theory

Action: Creates a subgraph from a given graph using either two integers or a law.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:subset** in-graph {subset-law | low-bounds up-bounds})

Arg Types:

in–graph	graph
subset–law	law
low–bounds	integer
up–bounds	integer

Returns: graph

Errors: None

Description: Given an ordered graph, a subgraph may be formed using one of two techniques. One method takes in two integers and the other takes a law pointer.

in–graph specifies a graph.

subset–law specifies a law. This extension with subset–law returns the set of all vertices such that their order evaluates as true along with the all edges that have both of their adjacent vertices evaluating as true orders.

This extension with low–bounds and up–bounds returns a subgraph in one of two ways.

1. If low–bounds<up–bounds, then the set of all vertices with orders between low–bounds and up–bounds is returned along with all edges that have both of their adjacent vertices in this set.
2. If up–bounds<low–bounds, then the set of all vertices with orders not between low–bounds and up–bounds is returned along with all edges that have both of their adjacent vertices in this set.

Limitations: None

Example:

```
; graph:subset
; Create a simple graph
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(graph:order-from g1 "a")
;; 4
(define g2 (graph:subset g1 1 3))
;; g2
(define g3 (graph:subset g1 "x>2"))
;; g3
(define law1 (law "(x>2)or(x=0)"))
;; law1
(define g4 (graph:subset g1 law1))
;; g4
```

graph:subtract

Scheme Extension: Graph Theory, Booleans

Action: Performs a Boolean subtract operation of two graphs.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:subtract** in-graph1 in-graph2 in-keep)

Arg Types:

in–graph1	graph
in–graph2	graph
in–keep	boolean

Returns: graph

Errors: None

Description: Refer to Action.

in–graph1 and in–graph2 specifies the graph.

The in–keep argument with a value true (#t) specifies that the edges going to common elements are kept.

Limitations: None

Example:

```
; graph:subtract
; Create some simple graphs.
(define g1 (graph "I-me me-myself myself-mine I-we
    we-us us-them"))
;; g1
(define g2 (graph "he-she it-thing they-those us-we
    them-us"))
;; g2
(define g3 (graph:subtract g1 g2 #f))
;; g3
(define g4 (graph:subtract g2 g1 #f))
;; g4
(define g5 (graph:subtract g2 g1 #t))
;; g5
```

graph:subtract–edges

Scheme Extension: Graph Theory

Action: Subtracts the edges of graph1 from graph2 returning the result.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: `(graph:subtract-edges in-graph1 in-graph2)`

Arg Types: in–graph1 graph
in–graph2 graph

Returns: graph

Errors: None

Description: Refer to Action.

in–graph1 and in–graph2 specifies the graph.

Limitations: None

Example:
```
; graph:subtract-edges
; Create a simple graph.
(define g1 (graph "a-b b-c c-e c-d c-f f-g f-h"))
;; g1
(define g2 (graph "c-f f-g f-h"))
;; g2
(define g3 (graph:subtract-edges g1 g2))
;; g3
```

graph:total–weight

Scheme Extension: Graph Theory

Action: Returns the total weight associated with the edges of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: `(graph:total-weight in-graph)`

Arg Types: in–graph graph

Returns: real

Errors: None

Description: If weights are assigned to individual edges of a graph, this returns the total weight for all of the edges.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:total-weight
; Create a simple graph.
(define g1 (graph ″a-b b-c c-e c-d c-f f-g f-h″))
;; g1
(graph:edge-weight g1 ″a″ ″b″ 3)
;; #[graph ″a-b b-c c-d c-e c-f f-g f-h″]
(graph:edge-weight g1 ″c-e″ 5)
;; #[graph ″a-b b-c c-d c-e c-f f-g f-h″]
(graph:total-weight g1)
;; 8
```

graph:tree?

Scheme Extension: Graph Theory

Action: Determines whether or not a given graph is a tree structure.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:tree?** in-graph)

Arg Types: in–graph graph

Returns: boolean

Errors: None

Description: Refer to Action.

in–graph specifies a graph.

Limitations: None

Example:

```
; graph:tree?
; Create a simple graph.
(define g1 (graph ″a-b b-c c-e c-d c-f f-g f-h″))
;; g1
(graph:tree? g1)
;; #t
(graph:linear? g1)
;; #f
(graph:cycle? g1)
;; #f
```

graph:unite

Scheme Extension: Graph Theory, Booleans

Action: Performs a Boolean unite operation of two graphs.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:unite** in-graph1 in-graph2)

Arg Types:

in–graph1	graph
in–graph2	graph

Returns: graph

Errors: None

Description: Given two graphs, this extension returns a new graph that is a Boolean union of the two.

in–graph1 and in–graph2 specifies the graph.

Limitations: None

Example:

```
; graph:unite
; Create some simple graphs.
(define g1 (graph "I-me me-myself myself-mine I-we
    we-us us-them"))
;; g1
(define g2 (graph "he-she it-thing they-those us-we
    them-us"))
;; g2
(define g3 (graph:unite g1 g2))
;; g3
```

graph:vertex–entities

Scheme Extension: Graph Theory

Action: Returns a list of entities that are associated with the vertices of a graph.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:vertex-entities** in-graph [use-ordering=#f])

Arg Types: in–graph — graph
use–ordering — boolean

Returns: (entity ...)

Errors: None

Description: A graph can be created using faces, cells, or wires, which become vertices of the graph.

in–graph specifies a graph.

If use–ordering is true (#t), sorts the result by graph order. The default value is false (#f).

Limitations: None

Example:
```
; graph:vertex-entities
; Create an example using entities.
(define e1 (edge:linear (position 10 10 0)
    (position 10 -10 0)))
;; e1
(define e2 (edge:linear (position 10 -10 0)
    (position -10 -10 0)))
;; e2
(define e3 (edge:linear (position -10 -10 0)
    (position -10 10 0)))
;; e3
(define e4 (edge:linear (position -10 10 0)
    (position 10 10 0)))
;; e4
(define g1 (graph (list e1 e2 e3 e4)))
;; g1
(graph:edge-entities g1)
;; (#[entity 5 1] #[entity 4 1] #[entity 3 1]
;; #[entity 2 1])
(graph:vertex-entities g1)
;; (#[entity 6 1] #[entity 7 1] #[entity 8 1]
;; #[entity 9 1] #[entity 10 1] #[entity 11 1]
;; #[entity 12 1] #[entity 13 1])
(define b1 (solid:block (position -5 -10 -20)
    (position 5 10 15)))
;; b1
(define faces1 (entity:faces b1))
;; faces1
; Turn the block faces into vertices of the graph.
```

```
(define g2 (graph faces1))
;; g2
(graph:edge-entities g2)
;; ()
(graph:vertex-entities g2)
;; (#[entity 20 1] #[entity 19 1] #[entity 18 1]
;; #[entity 17 1] #[entity 16 1] #[entity 15 1])
(define g3 (graph:unite g1 g2))
;; g3
(graph:edge-entities g3)
;; (#[entity 2 1] #[entity 3 1] #[entity 4 1]
;; #[entity 5 1])
(graph:vertex-entities g3)
;; (#[entity 13 1] #[entity 12 1] #[entity 11 1]
;; #[entity 10 1] #[entity 9 1] #[entity 8 1]
;; #[entity 7 1] #[entity 6 1] #[entity 15 1]
;; #[entity 16 1] #[entity 17 1] #[entity 18 1]
;; #[entity 19 1] #[entity 20 1])
```

graph:which–component

Scheme Extension: Graph Theory

Action: Returns the number of the component that a given graph element belongs to.

Filename: kern/kern_scm/graph_scm.cxx

APIs: None

Syntax: (**graph:which-component** in-graph in-object)

Arg Types:

in–graph	graph
in–object	string | entity

Returns: integer

Errors: None

Description: This extension is useful if the given in–graph has multiple components. It determines which component a given in–object is part of and returns its component number. The graph:component command then creates a new graph from just the elements of a single component.

in–graph specifies a graph.

in–object specifies a component. The component is selected by providing a string which is the name of an element of the component or an entity which is associated with an element of the component.

Limitations: None

Example:
```
; graph:which-component
; Create a simple example
(define g1 (graph "me-you us-them
    we-they them-they
    FIDO-SPOT SPOT-KING SPOT-PETEY"))
;; g1
; CAREFUL: The order of the graph output may
; not be the same each time.
(graph:components g1)
;; 3
(graph:which-component g1 "me")
;; 2
(define g2 (graph:component g1 2))
;; g2
(define g3 (graph:component g1 "me"))
;; g3
```

gvector

Scheme Extension: Mathematics

Action: Creates a new gvector given coordinates *x*, *y*, and *z*.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector** x y z [space=model])

Arg Types:

x	real
y	real
z	real
space	string

Returns: gvector

Errors: None

Description: Refer to Action.

x defines the *x*–coordinate relative to the active coordinate system.

y defines the *y*–coordinate relative to the active coordinate system.

z defines the *z*–coordinate relative to the active coordinate system.

The optional space argument defaults to "WCS". If no active WCS exists, space defaults to "model". The other optional space arguments return a gvector in the new coordinate system. The values for the space argument are:

- "wcs" is the default if an active WCS exists. Otherwise, the default is "model".
- "model" means that the *x*, *y*, and *z* values are with respect to the model. If the model has an origin other than the active WCS, this returns the position relative to the active coordinate system in rectangular Cartesian coordinates.
- "polar" or "cylindrical" mean that the *x*, *y*, and *z* values are interpreted as the radial distance from the *z*–axis, the polar angle in degrees measured from the *xz* plane (using right–hand rule), and the *z* coordinate, respectively. This returns the *x*, *y*, and *z* terms with respect to the active coordinate system.
- "spherical" means that the provided *x*, *y*, and *z* values are the radial distance from the origin, the angle of declination from the *z*–axis in degrees, and the polar angle measured from the *xz* plane in degrees, respectively. This returns the *x*, *y*, and *z* terms with respect to the active coordinate system.

Limitations: None

Example:

```
; gvector
; Create gvectors of various types.
(gvector 3 3 3)
;; #[gvector 3 3 3]
(gvector 5 5 5 "wcs")
;; #[gvector 5 5 5]
(gvector 5 5 5 "model")
;; #[gvector 5 5 5]
(gvector 5 5 5 "polar")
;; #[gvector 4.98097349045873 0.435778713738291 5]
(gvector 5 5 5 "cylindrical")
;; #[gvector 4.98097349045873 0.435778713738291 5]
(gvector 5 5 5 "spherical")
;; #[gvector 0.434120444167326 0.0379806174694798
;; 4.98097349045873]
```

gvector:+

Scheme Extension: Mathematics

Action: Adds two gvectors.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:+** gvector1 gvector2)

Arg Types:

gvector1	gvector
gvector2	gvector

Returns: gvector

Errors: None

Description: This extension returns the result of (gvector1 + gvector2) as a gvector.

gvector1 defines the first gvector.

gvector2 defines the second gvector.

Limitations: None

Example:

```
; gvector:+
; Add two gvectors by components.
(gvector:+ (gvector 1 3 2) (gvector 2 2 2))
;; #[gvector 3 5 4]
```

gvector:–

Scheme Extension: Mathematics

Action: Subtracts two gvectors.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:-** gvector1 gvector2)

Arg Types:

gvector1	gvector
gvector2	gvector

Returns: gvector

Errors: None

Description: This extension returns the result of (gvector1 – gvector2) as a gvector.

gvector1 defines the start location.

gvector2 defines the end location for both gvectors.

Limitations: None

Example:

```
; gvector:-
; Subtract two gvectors by components.
(gvector:- (gvector 1 3 2) (gvector 2 2 2))
;; #[gvector -1 1 0]
```

gvector:copy

Scheme Extension: Mathematics

Action: Creates a gvector by copying an existing gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:copy** gvector)

Arg Types: gvector gvector

Returns: gvector

Errors: None

Description: Refer to Action.

gvector specifies a gvector.

Limitations: None

Example:

```
; gvector:copy
; Create a gvector by copying an existing gvector.
(define copy (gvector:copy (gvector 6 5 2)))
;; copy
```

gvector:cross

Scheme Extension: Mathematics

Action: Gets the cross product of two gvectors.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:cross** gvector1 gvector2)

Arg Types:
gvector1 gvector
gvector2 gvector

Returns: gvector

Errors: None

Description: If the *i, j, k* components of vector *a* are <*a1*, *a2*, *a3*>, and the *i, j, k* components of vector *b* are <*b1*, *b2*, *b3*>, the cross product *a* x *b* is:

```
    |a2  a3  |       |a1  a3  |       |a1  a2  |
a x b = |        |i  -   |        |j  +   |        |k
        |b2  b3  |       |b1  b3  |       |b1  b2  |

a x b =        [(a2)(b3)-(b2)(a3)]i
             - [(a1)(b3)-(b1)(a3)]j
             + [(a1)(b2)-(b1)(a2)]k
```

The resulting cross product vector is perpendicular to both input vectors. The cross product *a x b* is not the same as the cross product *b x a*; they point in opposite directions (180 degrees from one another).

gvector1 specifies the first vector.

gvector2 specifies the second vector.

Limitations: None

Example:

```
; gvector:cross
; Compute the cross product of two gvectors.
(gvector:cross (gvector 2 2 2) (gvector 5 3 8))
;; #[gvector 10 -6 -4]
(gvector:cross (gvector 5 3 8) (gvector 2 2 2))
;; #[gvector -10 6 4]
```

gvector:dot

Scheme Extension: Mathematics

Action: Gets the dot product of two gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:dot** gvector1 gvector2)

Arg Types: gvector1 gvector
gvector2 gvector

Returns: real

Errors: None

Description: If the *i, j, k* components of vector *a* are <*a1*, *a2*, *a3*>, and the *i, j, k* components of vector *b* are <*b1*, *b2*, *b3*>, the dot product *a* . *b* is:

```
gvector1    =   (a1 a2 a3) = a
gvector2    =   (b1 b2 b3) = b
a . b       =   (a1*b1 + a2*b2 + a3*b3)
a . b       =   |a||b|cosq
                ; where q is the angle
                ; between a and b
```

The result of a dot product is a scalar value.

gvector1 specifies the first vector.

gvector2 specifies the second vector.

Limitations: None

Example:

```
; gvector:dot
; Compute the dot product of two gvectors.
(gvector:dot (gvector 3 5 1) (gvector 2 4 7))
;; 33
```

gvector:from–to

Scheme Extension: Mathematics

Action: Creates a gvector between two positions.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:from-to** position1 position2)

Arg Types: position1 position
position2 position

Returns: gvector

Errors: None

Description: This extension returns the gvector from position1 to position2.

position1 specifies the start location of the gvector.

position2 specifies the end location of the gvector.

Limitations: None

Example:
```
; gvector:from-to
; Create a gvector from one position to another.
(gvector:from-to (position 0 0 0) (position 5 1 6))
;; #[gvector 5 1 6]
```

gvector:length

Scheme Extension: Mathematics, Analyzing Models

Action: Gets the length of a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:length** gvector)

Arg Types: gvector gvector

Returns: real

Errors: None

Description: Returns the length of a gvector as a real value.

gvector specifies a gvector.

Limitations: None

Example:
```
; gvector:length
; Determine the length of two gvectors.
(gvector:length (gvector 0 6 0))
;; 6
(gvector:length (gvector 4 4 4))
;; 6.92820323027551
```

gvector:parallel?

Scheme Extension: Mathematics, Analyzing Models

Action: Determines if two gvectors are parallel.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:parallel?** gvector1 gvector2)

Arg Types:

gvector1	gvector
gvector2	gvector

Returns: boolean

Errors: None

Description: This extension returns #t if gvector1 and gvector2 are parallel; otherwise, it returns #f. A zero gvector is not parallel to anything, including itself, so it causes the extension to return #t.

gvector1 specifies the first vector.

gvector2 specifies the second vector.

Limitations: None

Example:

```
; gvector:parallel?
; Determine if two gvectors are parallel.
(gvector:parallel? (gvector 3 5 0) (gvector 6 10 0))
;; #t
(gvector:parallel? (gvector 1 0 0) (gvector 0 1 0))
;; #f
```

gvector:perpendicular?

Scheme Extension: Mathematics

Action: Determines if two gvectors are perpendicular.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:perpendicular?** gvector1 gvector2)

Arg Types:

gvector1	gvector
gvector2	gvector

Returns: boolean

Errors: None

Description: This extension returns #t if the gvectors are perpendicular; otherwise, it returns #f. A zero gvector is perpendicular to all gvectors, including itself, and it causes the extension to return #f.

gvector1 specifies the first vector.

gvector2 specifies the second vector.

Limitations: None

Example:

```
; gvector:perpendicular?
; Determine if two gvectors are perpendicular.
(gvector:perpendicular? (gvector 3 5 0)
    (gvector 6 10 0))
;; #f
(gvector:perpendicular? (gvector 1 0 0)
    (gvector 0 1 0))
;; #t
```

gvector:reverse

Scheme Extension: Mathematics

Action: Reverses the direction of a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:reverse** gvector)

Arg Types: gvector gvector

Returns: gvector

Errors: None

Description: Refer to Action.

gvector specifies a gvector.

Limitations: None

Example:

```
; gvector:reverse
; Reverses the direction of a gvector.
(gvector:reverse (gvector 0 1 0))
;; #[gvector 0 -1 0]
```

gvector:scale

Scheme Extension: Mathematics

Action: Multiplies a gvector by a scalar number to produce a new gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:scale** gvector scale)

Arg Types:

gvector	gvector
scale	real

Returns: gvector

Errors: None

Description: The resulting gvector is the original gvector scaled by the number.

gvector specifies the original gvector to be scaled by the scaling factor.

scale specifies the scaling factor.

Limitations: None

Example:

```
; gvector:scale
; Multiply two gvectors by a scaling factor.
(gvector:scale (gvector 0 -1 0) 3)
;; #[gvector 0 -3 0]
(gvector:scale (gvector 0 -1 0) -7)
;; #[gvector 0 7 0]
```

gvector:set!

Scheme Extension: Mathematics

Action: Sets a gvector's direction given components of *x*, *y*, and *z*.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:set!** gvector {x y z})

Arg Types:

gvector	gvector
x	real
y	real
z	real

Returns: gvector

Errors: None

Description: The coordinates are computed relative to the active coordinate system.

gvector specifies the original *x*–, *y*–, and *z*–components.

x specifies the value to replace the original *x*–value specified in gvector.

y specifies the value to replace the original *y*–value specified in gvector.

z specifies the value to replace the original *z*–value specified in gvector.

Limitations: None

Example:

```
; gvector:set!
; Set new x-, y-, and z-components
; in an existing gvector.
(define vector1 (gvector 1 0 0))
;; vector1
(gvector:set! vector1 0 7 3)
;; #[gvector 0 7 3]
vector1
;; #[gvector 0 7 3]
(define outline (gvector 0 0 1))
;; outline
(gvector:set! outline 3 5 4)
;; #[gvector 3 5 4]
```

gvector:set–x!

Scheme Extension: Mathematics

Action: Sets the *x*–direction component of a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:set-x!** gvector x)

Arg Types:

gvector	gvector
x	real

Returns: real

Errors: None

Description: The coordinates are computed relative to the active coordinate system. This extension returns the *x*–value as a real.

gvector specifies the original *x*–, *y*–, and *z*–values.

x specifies the value to replace the original *x*–value specified in gvector.

Limitations: None

Example:
```
; gvector:set-x!
; Set new x-, y-, and z-components
; in an existing gvector.
(define vector1 (gvector 1 0 0))
;; vector1
; Set a new x-component in an existing gvector.
(gvector:set-x! vector1 3)
;; 3
```

gvector:set–y!

Scheme Extension: Mathematics

Action: Sets the *y*–direction component of a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:set-y!** gvector y)

Arg Types:

gvector	gvector
y	real

Returns: real

Errors: None

Description: The coordinates are computed relative to the active coordinate system. This extension returns the *y*–value as a real.

gvector identifies the original *x*–, *y*–, and *z*–values.

y specifies the value to replace the original *y*–value specified in gvector.

Limitations: None

Example:
```
; gvector:set-y!
; Set new x-, y-, and z-components
; in an existing gvector.
(define vector1 (gvector 1 0 0))
;; vector1
; Set a new y-component in an existing gvector.
(gvector:set-y! vector1 6)
;; 6
```

gvector:set–z!

Scheme Extension: Mathematics

Action: Sets the *z*–direction component of a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:set-z!** gvector z)

Arg Types:

gvector	gvector
z	real

Returns: real

Errors: None

Description: The coordinates are computed relative to the active coordinate system. This extension returns the *z*–value as a real.

gvector identifies the original *x*–, *y*–, and *z*–values.

z specifies the value to replace the original *z*–value specified in gvector.

Limitations: None

Example:

```
; gvector:set-z!
; Set new x-, y-, and z-components
; in an existing gvector.
(define vector1 (gvector 1 0 0))
;; vector1
; Set a new z-component in an existing gvector.
(gvector:set-z! vector1 2)
;; 2
```

gvector:transform

Scheme Extension: Mathematics, Transforms

Action: Applies a transform to a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:transform** gvector transform)

Arg Types:

gvector	gvector
transform	transform

Returns: gvector

Errors: None

Description: Refer to Action.

gvector specifies the gvector to apply the transformation.

transform could be any valid transform.

Limitations: None

Example:

```
; gvector:transform
; Create a gvector.
(define vector1 (gvector 1 1 0))
;; vector1
; Apply a transform to a gvector.
(gvector:transform vector1
    (transform:reflection (position 0 0 0)
    (gvector 1 0 0)))
;; #[gvector -1 1 0]
```

gvector:unitize

Scheme Extension: Mathematics

Action: Creates a new gvector as a unit vector in the same direction as the specified gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:unitize** gvector)

Arg Types: gvector gvector

Returns: gvector

Errors: None

Description: Refer to Action.

gvector defines the vector to be unitized.

Limitations: None

Example:

```
; gvector:unitize
; Create a gvector.
(define vector1 (gvector 7 3 0 "model"))
;; vector1
; Create a gvector as a unit vector.
(gvector:unitize vector1)
;; #[gvector 0.919145030018058 0.393919298579168 0]
```

gvector:x

Scheme Extension: Mathematics

Action: Gets the *x*–component of a gvector relative to the active coordinate system.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:x** gvector)

Arg Types: gvector gvector

Returns: real

Errors: None

Description: This extension returns the *x*–coordinate of the gvector, transformed to the active WCS.

gvector specifies a gvector.

Limitations: None

Example:

```
; gvector:x
; Create a gvector.
(define vector1 (gvector 7 5 0 ”spherical”))
;; vector1
; Determine the x-component of a gvector.
(gvector:x vector1)
;; 0.610090199233607
```

gvector:y

Scheme Extension: Mathematics

Action: Gets the *y*–component of a gvector relative to the active coordinate system.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:y** gvector)

Arg Types: gvector gvector

Returns: real

Errors: None

Description: This extension returns the *y*–coordinate of the gvector, transformed to the active WCS.

gvector specifies a gvector.

Limitations: None

Example:

```
; gvector:y
; Create a gvector.
(define vector1 (gvector 7 5 0 "spherical"))
;; vector1
; Determine the y-component of a gvector.
(gvector:y vector1)
;; 0
```

gvector:z

Scheme Extension: Mathematics

Action: Gets the *z*–component of a gvector relative to the active coordinate system.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector:z** gvector)

Arg Types: gvector gvector

Returns: real

Errors: None

Description: This extension returns the *z*–coordinate of the gvector, transformed to the active WCS.

gvector specifies a gvector.

Limitations: None

Example:

```
; gvector:z
; Create a gvector.
(define vector1 (gvector 7 5 0 "spherical"))
;; vector1
; Determine the z-component of a gvector.
(gvector:z vector1)
;; 6.97336288664222
```

gvector?

Scheme Extension: Mathematics

Action: Determines if a Scheme object is a gvector.

Filename: kern/kern_scm/gvec_scm.cxx

APIs: None

Syntax: (**gvector?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: Refer to Action.

object specifies the scheme–object that has to be queried for a gvector.

Limitations: None

Example:

```
; gvector?
; Create a gvector.
(define vector1 (gvector 7 5 0 "spherical"))
;; vector1
; Determine if the following objects are gvectors.
(gvector? vector1)
;; #t
(gvector? (position 0 0 0))
;; #f
(gvector? -4)
;; #f
```

history:ensure–empty–root–state

Scheme Extension: History and Roll

Action: Adds empty delta state to the beginning of the history stream so that users can roll to a state with no entities.

Filename: kern/kern_scm/hist_scm.cxx

APIs: api_ensure_empty_root_state, api_get_state_id

Syntax: (**history:ensure-empty-root-state** [history])

Arg Types: history history

Returns: integer

Errors: None

Description: This routine examines the root delta state of the specified history stream. If the root state is empty (no bulletin boards), then it does nothing. If the root state is not empty, then it adds a new, empty, root state immediately *before* the original root state. In either case, it returns the ID number of the (empty) root state.

history specifies a history stream.

Limitations: None

Example:
```
; history:ensure-empty-root-state
; No example available at this time
```

history:get–active–state–id

Scheme Extension: History and Roll

Action: Returns an integer representing the active state.

Filename: kern/kern_scm/hist_scm.cxx

APIs: api_get_active_state, api_get_state_id, api_note_state

Syntax: (**history:get-active-state-id** [history])

Arg Types: history history

Returns: integer

Errors: None

Description: Returns an integer representing the active state's id in the active history stream. An optional history stream may be specified, causing its associated active state to be returned. If no stream is specified, the default history stream is used.

history specifies a history stream.

Limitations: None

Example:
```
; history:get-active-state-id
; Example not available for this release
```

history:get–default

Scheme Extension: History and Roll

Action: Returns the default history stream.

Filename: kern/kern_scm/hist_scm.cxx

APIs: api_get_default_history

Syntax: (**history:get-default**)

Arg Types: none

Returns: integer

Errors: None

Description: Refer to Action.

Limitations: None

Example:

```
; history:get-default
; get the default history stream
(history:get-default)
;; #[(deleted) history -1]
```

history:get–entity–from–id

Scheme Extension: History and Roll

Action: Returns an ENTITY from a given tag id.

Filename: kern/kern_scm/hist_scm.cxx

APIs: api_get_entity_from_id

Syntax: (**history:get-entity-from-id** id [history])

Arg Types:

id	integer
history	history

Returns: integer

Errors: The id must be valid.

Description: Returns an ENTITY from a given tag id in the HISTORY_STREAM specified. If no stream is specified, the default stream is used.

id specifies an entity identifier.

history specifies a history stream.

Limitations: None

Example:

```
; history:get-entity-from-id
; Create a block
(define b (solid:block (position -10 -10 -10)
    (position 10 10 10)))
;; b
(define lop (lop:offset-body b 5))
;; lop
(define f (pick:face (ray (position 0 0 0)
    (gvector 1 0 0))))
;; f
(entity:set-color f BLUE)
;; ()
(define id (entity:get-id f))
;; id
(roll)
;; -1
(roll)
;; -1
(entity:set-color (history:get-entity-from-id
    id) RED)
;; ()
```

history:validate–streams

Scheme Extension: History and Roll

Action: Checks all history streams for validity.

Filename: kern/kern_scm/hist_scm.cxx

APIs: api_check_histories

Syntax: (**history:validate-streams**)

Arg Types: None

Returns: boolean

Errors: None

Description: Checks all history streams for mixing and bad entity ids. Returns #t if all are OK, or #f otherwise (also reports error to debug_file_ptr).

Limitations: None

Example:

```
; history:validate-streams
;; make a stream
(define block (solid:block 0 0 0 10 10 10))
;; block
(roll)
;; -1
(define sphere (solid:sphere 0 0 0 10))
;; sphere
; verify that it (and all other streams) are valid
(history:validate-streams)
; 1 history streams checked.
;; #t
```