Chapter 14.
Scheme Extensions Ia thru Qz

Topic: Ignore

is:helix

Scheme Extension: Laws, Space Warping

Action: Determines whether or not the input entity is a helix.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(is:helix in–entity)`

Arg Types: in–ent entity

Returns: boolean

Errors: None

Description: The input entity in–ent should be an edge.

in–entity specifies an entity.

Limitations: None

Example:

```
; is:helix
; Create a profile to sweep into a helix.
(option:set ”sil” #f)
;; #t
(define profile1 (edge:linear
    (position 5 0 0) (position 10 0 0)))
;; profile1
(is:helix profile1)
;; #f
(define path1(edge:helix
    (position 0 0 0)
    (position 0 40 0)
    (gvector 1 0 0) 10 20 #t))
;; path1
(define helix1(sweep:law profile1 path1))
;; helix1
(define edge–list (entity:edges helix1))
;; edge–list
(is:helix (list–ref edge–list 0))
;; #f
(is:helix (list–ref edge–list 1))
; Axis 0.000000 1.000000 0.000000
; Root 0.000000 9.325620 0.000000
; Pitch 20.000000
; Radius 5.000000
; Right Handed
;; #t
(is:helix (list–ref edge–list 2))
;; #f
(is:helix (list–ref edge–list 3))
; Axis 0.000000 1.000000 0.000000
; Root 0.000000 9.328920 0.000000
; Pitch 20.000000
; Radius 10.000000
; Right Handed
;; #t
```

law

Scheme Extension: Laws

Action: Creates a law Scheme data type composed of one or more law functions.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_str_to_law

Syntax: `(law {"law-functions" | type} [law-data]*)`

Arg Types:		
	"law–functions"	string
	type	law | real | position | par–pos | gvector
	law–data	law | real | position | par–pos | gvector | transform | edge | wire

Returns: law

Errors: None

Description: The law Scheme extension creates a law Scheme data type composed of one or more law functions.

"law–functions" is a string specifying law function enclosed in quotation marks.

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

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

The optional argument defined in law–data must agree with the items specified within law–functions in type, number, and order. For example, if the law–functions calls for law1, edge2, and wire3, the input must be a list with at least three items: the first item must be a law; the second item must be an edge; and the third item must be a wire. The index number of the item within the law function specifies its ordering in the input list.

position, par–pos, or gvector may be used as law–data, which correspond to the law# and are converted into the "vec" law function.

Limitations: None

Example:

```
; law
; Create a new law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
; Evaluate the given law at 1.5 radians
(law:eval law1 1.5)
;; 3.6792627983323
```

```
; Another example using law data
; This creates a law that returns the position
; of an edge with its position mapped to the
; closed interval [0,1].
; Create an edge.
(define edge1 (edge:linear (position 0 0 0)
    (position 40 40 0)))
;; edge1
(define edge–law (law
    "map(cur(edge1)),edge1)" edge1))
;; edge–law
```

law:bounds

Scheme Extension: Laws, Debugging

Action: Finds the bounds for a law function.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:bounds** law1 [low high])

Arg Types:

law1	law
low	real
high	real

Returns: (real ...)

Errors: Checking the bounds on a law with an infinite domain returns an error.

Description: This returns a list of bounds of the law from the different interval pairs entered.

law1 specifies a law.

low and high are optional intervals over which the bounds are computed. If not specified, the extension attempts to compute the domain from the law itself.

Limitations: None

Example:

```
; law:bounds
; Define a law and find the bounds on interval.
; Create a law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
(law:bounds law1 -2 20)
; -5.350134 to 423.799265
; Tolerance: 0.001000
;; (-5.35013412467261 423.799264988314 0.001)
```

law:check

Scheme Extension: Laws, Debugging

Action: Prints out the laws string.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:check** law)

Arg Types:

law	law | string | real | position | par–pos | gvector

Returns: string

Errors: None

Description: Prints out the laws string or a simplified laws string. Determines if a law is zero, constant, or linear. Tells what its takes and returns. Return the domain of the law.

law specifies a law.

Limitations: None

Example:

```
; law:check
; Create a law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
; Check the law.
(law:check law1)
; (X+X^2)-COS(X
; Simplified (X+X^2)-COS(X)
; Zero #f
; Constant #f
; Linear #f
; Takes 1 argument
; Returns 1 argument
; Dx = 1+2*X+SIN(X)
; Inverse of this law is not available.
; Numerator = (X+X^2)-COS(X)
; Denominator = 1
; The domain in variable 0 is infinite
;; #t
```

law:count

Scheme Extension: Laws, Debugging

Action: Gets a count of laws in use.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:count**)

Arg Types: None

Returns: boolean

Errors: None

Description: A static list holding each of one law class is maintained. Therefore, the law count number will never be zero. This is used for testing of memory leaks in laws. This command should be used with the collect Scheme extension to find memory leaks.

Limitations: None

Example:

```
; law:count
; Create a law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
; Check to see how many laws are in use.
(law:count)
;; 121
; This return depends on how many laws were defined
; and the number of laws maintained in the law static
; list.
```

law:cubic

Scheme Extension: Laws, Mathematics

Action: Creates a cubic law given {a,b,f(a),f(b),f'(a),f'(b)}.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_make_cubic

Syntax: `(law:cubic a b f-a f-b df-a df-b)`

Arg Types:

a	real
b	real
f–a	real
f–b	real
df–a	real
df–b	real

Returns: boolean

Errors: None

Description: Produces a cubic a polynomial with given boundary conditions for both it and its first derivative.

a and b specifies the boundary values.

f–a and f–b are the desired output of the law at a and b.

df–a and df–b are the desired output of the first derivative at a and b.

Limitations: None

Example:

```
; law:cubic
; Create a law.
(define law1 (law:cubic 0 1 2 3 4 5))
;; law1
; According to the definition, if a 0 is input,
; the output should be 2 and its first derivative
; at 0 is 4.
(law:eval law1 0)
;; 2
(law:eval law1 1)
;; 3
(define deriv (law:derivative law1))
;; deriv
(law:eval deriv 0)
;; 4
(law:eval deriv 1)
;; 5
```

law:derivative

Scheme Extension: Laws

Action: Creates a law object that is the derivative of the given law with respect to the given variable.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:derivative** law [with–respect–to])

Arg Types:

law	law | string | real | position | par–pos | gvector
with–respect–to	law | string

Returns: law

Errors: None

Description: The law:derivative returns the exact derivative of the given law function.

law specifies a law.

with–respect–to is an optional argument specifying an identity law, such as x, y, z, or a#, etc. If with–respect–to is provided, then the derivative is performed with respect to this argument; else the derivative is assumed to be with respect to x or a1.

Limitations: None

Example:

```
; law:derivative
; Define a law to use.
(define law1 (law "x^2"))
;; law1
; To get exact results, take the exact derivative.
; Create the derivative of that law.
(define d-law (law:derivative law1))
;; d-law
; Evaluate the derivative.
(law:eval d-law 3)
;; 6
; Evaluate the second derivative.
(law:eval (law:derivative d-law) 3)
;; 2
; Find the numerical derivative at 3.
(law:nderivative law1 3 "x" 1)
;; 5.9999999999838
; Find the second numerical derivative at 3.
(law:nderivative law1 3 "x" 2)
;; 1.9999997294066
```

law:end

Scheme Extension: Laws

Action: Returns the end of the domain for a law.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:end** in-law [term])

Arg Types:

in–law	law
term	integer

Returns: real

Errors: None

Description: Returns the end of the domain of a law.

in–law specifies a law.

The optional term argument could be used to localize which term in the law is used.

Limitations: None

Example:
```
; law:end
; Define a profile to sweep for interesting shape
(define profile1 (wire–body:points
    (list (position 5 0 0)
    (position 10 0 0)
    (position 10 5 0)
    (position 5 5 0)
    (position 5 0 0))))
;; profile1
(define path1(edge:helix
    (position 0 0 0)
    (position 0 40 0)
    (gvector 1 0 0) 10 20 #t))
;; path1
(define helix1(sweep:law profile1 path1))
;; helix1
(define edge–list(entity:edges helix1))
;; edge–list
(define law1
    (law "cur(EDGE1)"
    (list–ref edge–list 3)))
;; law1
(law:start law1)
;; 0
(law:end law1)
;; 12.5663706143592
```

law:eval

Scheme Extension: Laws, Analyzing Models

Action: Evaluates a given law or law expression with the specified input and returns a real or list of reals.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:eval** law input)

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	input	(entity | entity ...) | real | gvector | position | par–pos

Returns: real | (real ...)

Errors: None

Description: This Scheme extension evaluates the law at the given input value(s).

law can be a law Scheme type or a string enclosed by double quotation marks. law can be any combination of law functions. law can be multidimensional in domain and range. The vec law symbol can be used as part of the law definition to generate a multidimensional range (e.g., output is a list of reals).

input specifies the value(s) for evaluating the law.

Limitations: None

Example:

```
; law:eval
; Create a new law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
; Evaluate the given law at 1.5 radians
(law:eval law1 1.5)
;; 3.6792627983323
; Create a law with a multidimensional range.
(define mlaw (law "vec(x+2,x+3,x+4,x+5)"))
;; mlaw
(law:eval mlaw 3)
;; (5 6 7 8)
; Create a law with a multidimensional domain.
(define law2 (law "x^2 + y + z"))
;; law2
(law:eval law2 (list 2 3 4))
;; 11
(define law3 (law "a3^2 + a2 + a4"))
;; law3
; When evaluating this, the law function has 4 as its
; highest input index. Thus, the input list has to
; have at least four items. Input a1 is not used in
; the law function, so the first element of list
; is not used.
(law:eval law3 (list 1 2 3 4))
;; 15
; a3=3, a2=2, and a4=4. (a1=1).
```

law:eval–par–pos

Scheme Extension: Laws, Analyzing Models

Action: Evaluates a parameter position.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:eval–par–pos in–law input)`

Arg Types:

in–law	law | string | real | position | par–pos | gvector
input	(entity | (entity ...)) | real | gvector | position | par–pos

Returns: position | par–pos

Errors: The in–law must return 2 or 3 values.

Description: Refer to Action.

law can be a law Scheme type or a string enclosed by double quotation marks. law can be any combination of law functions.

input specifies the value(s) for evaluating the law.

Limitations: None

Example:

```
; law:eval–par–pos
; Evaluate a parameter's position
(define law1 (law "vec(x,x^2)"))
;; law1
(law:eval–par–pos law1 2)
;; #[par–pos 2 4]
(define law2 (law "vec(x^2, y+3*z)"))
;; law2
(law:eval–par–pos law2 (list 2 3 4))
;; #[par–pos 4 15]
```

law:eval–position

Scheme Extension: Laws, Analyzing Models

Action: Evaluates a given law or law expression with the specified input and returns a position or a par–pos.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:eval-position law input)`

Arg Types:

law	law | string | real | position | par–pos | gvector
input	(entity | (entity ...)) | real | gvector | position | par–pos

Returns: position | par–pos

Errors: The law must return 2 or 3 values.

Description: This Scheme extension evaluates the law at the given input. It is the same as law:eval except that it returns either a position or a par–pos instead of a real or list of reals. The output dimension of law has to be two (2) or three (3). The result is a par–pos when the output dimension is two, and position when it is three.

law can be a law Scheme type or a string enclosed by double quotation marks. law can be any combination of law functions.

input specifies the value(s) for evaluating the law.

Limitations: None

Example:

```
; law:eval–position
; Create a new law.
(define law1 (law ”vec(x,x^2,–cos(x))”))
;; law1
; Evaluate the given law at 1.5 radians
(law:eval–position law1 2)
;; #[position 2 4 0.416146836547142]
(define law2 (law ”vec(x^2, y, 3 * z)”))
;; law2
; Evaluate the second law at some input values.
(law:eval–position law2 (list 2 3 4))
;; #[position 4 3 12]
(define law3 (law ”vec(a3^2,a2,a4)”))
;; law3
; When evaluating this, the law function has 4 as its
; highest input index. Thus, the input list has to
; have at least four items. In law3, input A1 is
; not used in the law function, so the first element
; of list is not used.
; a3=3, a2=2, and a4=4. (a1=1).
(define law4 (law ”vec(a3^2,a4)”))
;; law4
(law:eval–position law3 (list 1 2 3 4))
;; #[position 9 2 4]
(law:eval law3 (list 1 2 3 4))
;; (9 2 4)
(law:eval–position law4 (list 1 2 3 4))
;; #[par–pos 9 4]
```

law:eval–vector

Scheme Extension: Laws, Analyzing Models

Action: Evaluates a given law or law expression with the specified input and returns a gvector.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:eval–vector** law input)

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	input	(entity | entity ...)) | real | position | par–pos

Returns: unspecified

Errors: The law must return 3 values.

Description: This Scheme extension evaluates the law at the given input. It is the same as law:eval except that it returns a gvector instead of a real or list of reals. The output dimension of law has to be three (3).

law can be a law Scheme type or a string enclosed by double quotation marks. law can be any combination of law functions.

input specifies the value(s) for evaluating the law.

Limitations: None

Example:

```
; law:eval-vector
; Create a new law.
(define law1 (law "vec(x,x^2,-cos(x))"))
;; law1
; Evaluate the given law at 1.5 radians
(law:eval-vector law1 2)
;; #[gvector 2 4 0.416146836547142]
(define law2 (law "vec(x^2, y, 3 * z)"))
;; law2
; Evaluate the second law at some input values.
(law:eval-vector law2 (list 2 3 4))
;; #[gvector 4 3 12]
(define law3 (law "vec(a3^2,a2,a4)"))
;; law3
; When evaluating this, the law function has 4 as its
; highest input index. Thus, the input list has to
; have at least four items. Input A1 is not used in
; the law function, so the first element of list
; is not used.
; a3=3, a2=2, and a4=4. (a1=1).
(law:eval-vector law3 (list 1 2 3 4))
;; #[gvector 9 2 4]
(law:eval law3 (list 1 2 3 4))
;; (9 2 4)
```

law:hedgehog

Scheme Extension: Laws, Sweeping

Action: Creates visible vector field lines along an edge or wire.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_hedgehog

Syntax:
```
(law:hedgehog
    {path vector–list [number–hair [size]]} |
    {hair–law base–law start–x end–x
    {[number–hair] | [start–y end–y [number–hair]] |
    [start–y end–y start–z end–z [number–hair]]}})
```

Arg Types:	
path	entity
vector–list	law | (law ...)
number–hair	integer
size	real
hair–law	law | string | gvector
base–law	law | string | position
start–x	real
end–x	real
start–y	real
end–y	real
start–z	real
end–z	real

Returns: boolean

Errors: None

Description: The law:hedgehog extension defines vector fields that are attached to a curve. They are useful for defining orientation when sweeping along a non–planar path. There are two major ways to use this command. The determination as to which method is used is based on the first argument type.

If the first argument is an entity, as would be the case for a wire body or edge, then that argument is assumed to be the path and should be followed by a Scheme list of laws, vector–list Optionally, the number of hairs on the hedgehog and its size can be specified.

If the first argument is not an entity, then it is assumed to be a hair–law, which can be a law, string, or gvector. hair–law specifies the direction for the hair vectors. This is followed by base–law., which specifies what is to receive the vector field. Thereafter, this can be followed by 0 or up to 5 arguments. number–hair specifies the number of vectors to use. The start and end parameters limit the range for displaying the vector field.

Limitations: None

Example:

```
; law:hedgehog
; The edge is not required, but is useful
; for visualization purposes. It is defined
; the same as base–law.
(define edge1
    (edge:law ”vec(cos(x),sin(x),x/10)” 0 50))
;; edge1
(define base–law
    (law ”vec(cos(x),sin(x),x/10)”))
;; base–law
(define vector–law (law ”vec(cos(x),sin(x),0)”))
;; vector–law
(define hedgehog1 (law:hedgehog vector–law
    base–law 0 50 200 ))
;; hedgehog1

; Another example.
; Clear out the result of the previous example
(part:clear)
;; #t
; The face is not required, but is useful
; for visualization purposes. It is defined
; the same as base–law.
(define face1
    (face:law ”vec(x,y,cos(x)*sin(y))” –5 5 –5 5))
;; face1
(define base–law1(law ”vec(x,y,cos(x)*sin(y))”))
;; base–law1
(define dx (law:derivative base–law))
;; dx
(define dy(law:derivative base–law1 ”y”))
;; dy
(define norm (law ”cross(law1,law2)” dx dy))
;; norm
(define hedgehog2 (law:hedgehog norm
    base–law –5 5 –5 5 20 ))
;; hedgehog2
```

```
; Another example.
; Clear out the previous example
(part:clear)
;; #t
(dl–item:erase hedgehog2)
;; ()
(define base–law2 (law "vec(x,y,z)"))
;; base–law2
(define hair–law (law "norm(vec(x,y,z))"))
;; hair–law
(define hedgehog3 (law:hedgehog hair–law
    base–law –10 10 –10 10 –10 10))
;; hedgehog3
; Clean up the DL items.
;(dl–item:erase hedgehog3)
; ()
```

law:inverse

Scheme Extension: Laws, Space Warping

Action: Returns a law that is the inverse of the supplied law.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:inverse** in–law)

Arg Types:

in–law	law

Returns: law

Errors: None

Description: The inverse of a given law is not supported for most laws. The only supported law having an inverse is something derived from the bend law.

in–law specifies a law.

Limitations: None

Example:

```
; law:inverse
; Create a cylinder
(define a–part (solid:cylinder (position –20 0 –2)
    (position 20 0 –2) 0.75))
;; a–part
(entity:bend a–part (position –8 0 –2)
    (gvector 0 0 1) (gvector 0 1 0) 4 90)
;; #[entity 2 1]
(define e–list (entity:edges a–part))
;; e–list
(define law1 (law "cur(EDGE1)"
    (list–ref e–list 0)))
;; law1
(define law2 (law:inverse law1))
;; law2
```

law:line–line–intersect

Scheme Extension: Laws

Action: Returns TRUE if two lines intersect.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:line–line–interset** point1 gvector1 point2 gvector2)

Arg Types:

point1	position
gvector1	gvector
point2	position
gvector2	gvector

Returns: boolean

Errors: None

Description: Returns TRUE if the line given by point1 + t * gvector1 intersects the line point2 + s * gvector2. Also prints out the parameter values of the two nearest nearest points.

point1 specifies a position on the first line.

gvector1 specifies the direction of the first line.

point2 specifies a position on the second line.

gvector2 specifies the direction of the second line.

Limitations: None

Example:

```
; law:line-line-intersect
; Create a cylinder
; intersect at (1 2 3)
(law:line-line-intersect (position 4 4 4)
    (gvector 3 2 1) (position 1 10 3)
    (gvector 0 -.3 0))
; Lines intersect at (1.000000 2.000000
;       3.000000)
; Param values are:
;       param0 at -1.000000
;       param1 at 26.666667
;; #t
; no intersection
(law:line-line-intersect (position 5 4 4)
    (gvector 3 2 1) (position 3 10 1)
    (gvector 0 -.3 0))
; Param values are:
;       param0 at -0.900000
;       param1 at 26.000000
;; #f
```

law:linear

Scheme Extension: Laws, Mathematics

Action: Creates a linear law given {a,b,f(a),f(b)}.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_make_linear

Syntax: (**law:linear** a b f-a f-b)

Arg Types:

a	real
b	real
f–a	real
f–b	real

Returns: boolean

Errors: None

Description: Produces a linear polynomial using the input boundary conditions.

a and b specifies the boundary values.

f–a and f–b specifies the desired output of the law at a and b.

Limitations: None

Example:
```
; law:linear
; Create a law.
(define law1 (law:linear 0 1 2 3))
;; law1
(law:eval 0)
;; 0
(law:eval 1)
;; 1
```

law:make–entity

Scheme Extension: Laws, Model Topology

Action: Creates a law entity data type composed of one law function.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_law_to_entity

Syntax:
```
(law:make–entity {some–law–functions | type}
    [law–data]*)
```

Arg Types:

some–law–functions	string
law–data	law | real | position | par–pos | gvector | transform | edge | wire
type	law | real | position | par–pos |

Returns: law

Errors: None

Description: This routine creates a law entity which can be saved and restored from a SAT file. The law does not have to be attached to model geometry.

Limitations: None

Example:
```
; law:make–entity
; Create a law to use
(define law1 (law "vec(x^2,cos(x)+arctan(y))"))
;; law1
(define law–ent (law:make–entity law1))
;; law–ent
```

law:make–rails

Scheme Extension: Laws, Sweeping

Action: Creates a rail law or a list of rail laws for use by sweeping.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_make_rails

Syntax:
```
(law:make-rails path [twist [axis [faces
    [user-rails [version]]]]])
```

Arg Types:

path	edge | wire
twist	law
axis	law | (law ...)
faces	face | (face ...)
user–rails	law | (law ...)
version	version_tag

Returns: law | (law ...)

Errors: None

Description: This produces a rail law or a list of rail laws that can be used by sweeping or law:hedgehog.

If the path is a single edge or a wire with a single underlying edge, then a single rail law is produced. Otherwise, the extension creates multiple rail laws, one for each underlying edge in the path.

The only required argument is the path. If no other arguments are supplied, then the following default rails are created:

– If the path is planar, the rail law is the planar normal.
– If the path is a helix, the rail law points towards the axis.
– If all edges in the wire are planar, then an array of rail laws is created, whereby each law in the array corresponds to an edge in the wire. The rail laws correspond to the planar normal of edges.
– If the path isn't one of the above cases, the rail uses minimum rotation.

Because the input arguments are chained together, "NULL" arguments (double quotation marks required) must be supplied for early elements in the argument list in order to change later arguments. If the input path is composed of multiple pieces, such as a wire with more than one underlying edge, then list arguments must supply the same number of elements as the number of path elements. They may pad their list with "NULL" arguments (double quotation marks required).

For example, assume that an input path has three underlying edges. Assume that a list of axis laws are to be supplied, but no twist law. The list of axis laws would have to contain three elements. The list of axis laws could contain “NULL”, which would result in the default rail for those corresponding portions of the path. The example call might look like:

```
(law:make-rails my-path "NULL"
    (list "NULL" my-law "NULL"))
```

The twist argument works on the whole rail. After the other rail parameters have been input and calculated, the law provided by twist operates on the whole set of rails. This takes in an angle of twist per distance along the path. The twist argument can be “NULL”.

The axis argument is used for path segments that have an implied center axis. An example of this might be a helix, an expanding helix, or the coil of a telephone handset cable. The axis argument is the derivative of the implied center axis, which tells the implied axis direction. When the axis is supplied, the created rails point towards it. The axis list can be padded with “NULL” for sections of the path that do not have an implied axis.

The face argument is used when a portion of the path segments borders a non–analytic face. The coedge of the wire provided as path must actually belong to the face entity supplied. The face must be non–analytic. The resulting rail is oriented to the face normal. The face list can be padded with “NULL” for sections of the path that do not have such a face.

The user–rails argument permits any default rail for a given section of the path to be overridden by the user–supplied law in the list. The user–rails list can be padded with “NULL” for sections of the path that are to use the default.

The version argument permits the user to specify a versiontag leaving the min_rotation_law as the rail. This behavior is not recommended for performance and robustness.

Limitations: None

Example:

```
; law:make-rails
; is:helix
(define path1 (wire-body:offset (wire-body
    (edge:linear (position 0 0 0) (position 0 0 20)))
    5 "x"))
;; path1
(define rail-1 (law:make-rails path1))
;; rail-1
; #[law
; "NORM(CROSS(DOMAIN(VEC(-(5*COS(X)),
; -(5*SIN(X)),1),0,20),VEC(0,0,1)))"]
; IF YOU WANT TO SEE THE RAILS, use law:hedgehog
```

law:nderivative

Scheme Extension: Laws, Analyzing Models

Action: Computes the numerical derivative of a law function with respect to a given variable, a given number of times.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_ndifferentiate_law

Syntax:

```
(law:nderivative law value [with-respect-to
    [number-of [type=0]]])
```

Arg Types:

law	law | string | real | position | par–pos | gvector
value	(entity | (entity ...)) | real | position | gvector | par–pos
with–respect–to	law
number–of	integer
type	integer

Returns: real

Errors: None

Description: The law:nderivative returns the numerical derivative of the given law function at the specified value, value.

law specifies a law.

value specifies a value at which the derivative is evaluated.

with–respect–to is an optional argument specifying an identity law, such as x, y, z, or a#, etc. If with–respect–to is provided, then the derivative is performed with respect to this argument; else the derivative is assumed to be with respect to x or a1.

If number–of is specified as some integer, then the extension performs multiple derivatives. If number–of is not specified, it is assumed to be 1.

type defines the type of derivative. Zero is normal (default); 1 is from the left; and 2 is from the right.

Limitations: None

Example:

```
; law:nderivative
; Define a law to use.
(define law1 (law "x^2"))
;; law1
; Find the numerical derivative at 3.
(law:nderivative law1 3 "x" 1)
;; 5.9999999999838
; Find the second numerical derivative at 3.
(law:nderivative law1 3 "x" 2)
;; 1.9999997294066
; To get exact results, take the exact derivative.
; Create the derivative of that law.
(define dlaw (law:derivative law1))
;; dlaw
; Evaluate the derivative.
(law:eval dlaw 3)
;; 6

; Example of left and right derivatives.
(law:nderivative "abs(x)" 0 "x" 1 1)
;; –1
(law:nderivative "abs(x)" 0 "x" 1 2)
;; 1
(law:nderivative "abs(x)" 0 "x" 1 0)
;; –1.52465930505774e–16
; In the case of non–differentiable functions,
; the normal option returns the average value.
```

law:nintegrate

Scheme Extension: Laws, Analyzing Models

Action: Computes the numerical integral of the given law function over the specified range.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_integrate_law

Syntax: `(law:nintegrate law low–lim up–lim [tol])`

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	low–lim	real
	up–lim	real
	tol	real

Returns: real

Errors: None

Description: The law:nintegrate Scheme extension computes a numerical integration on law.

law can be a defined law, a law string, or a constant number.

low–lim and up–lim are two real values that specify the lower and upper limit for integration.

tol is an optional real value that specifies the tolerance for the integration calculation.

Limitations: None

Example:

```
; law:nintegrate
; Define a law to integrate.
(define law1 (law "x + 1"))
;; law1
; Evaluate the law over the range 0 and 1.
(law:nintegrate law1 0 1)
;; 1.5
```

law:nmax

Scheme Extension: Laws, Analyzing Models

Action: Computes where a law is the maximum over a given interval.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:nmax law low–lim up–lim)`

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	low–lim	real
	up–lim	real

Returns: real

Errors: None

Description: law:nmax computes where a law is the maximum over a given interval.

law can be a defined law, a law string, or a constant number.

low–lim and up–lim are two real values that specify the interval.

The example defines law1 to be the law that returns the curvature of testcur1 with its parameter range mapped to [0,1]. The “CurC” law returns the curvature of its curve. Because the radius of curvature is the reciprocal of the curvature, finding where law1 is maximum finds where the radius of curvature is minimum, or where the tightest turn in the curve is located.

Limitations: None

Example:

```
; law:nmax
; Define a list of points for a spline edge
(define plist (list (position 0 0 0)
    (position 20 20 0) (position 40 0 0)
    (position 60 25 0) (position 40 50 0)
    (position 20 30 0) (position 0 50 0)))
;; plist
(define start (gvector 1 1 0))
;; start
(define end (gvector –1 1 0))
;; end
(define testcur1 (edge:spline plist start end))
;; testcur1
(define law1
    (law ”map(Curc(edge1),edge1)” testcur1))
;; law1
(define maxpoint (law:nmax law1 0 1))
;; maxpoint
; All curves in Scheme are parameterized from 0 to 1.
(define point1 (dl–item:point
    (curve:eval–pos testcur1 maxpoint)))
;; point1
(dl–item:display point1)
;; ()
; OUTPUT Result
```

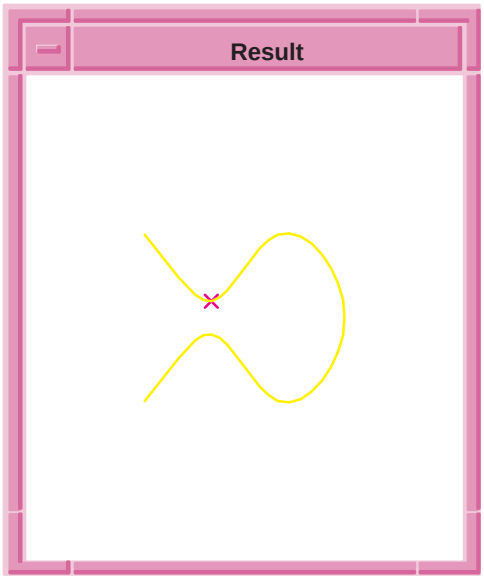

Figure 14-1. law:nmax – First Maximum Point on a Test Curve

law:nmin

Scheme Extension: Laws, Analyzing Models

Action: Computes where a law is the minimum over a given interval.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:nmin law low–lim up–lim)`

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	low–lim	real
	up–lim	real

Returns: real

Errors: None

Description: law:nmin computes where a law is the minimum over a given interval.

law can be a defined law, a law string, or a constant number.

low–lim and up–lim are two real values that specify the interval.

Limitations: None

Example:

```
; law:nmin
; Define a simple law to find minimum.
(law:nmin "x^2+x" –1 1)
;; –0.5

; Another Example.
; Create the points for a test curve.
(define plist (list (position 0 0 0)
    (position 20 20 20) (position –20 20 20)
    (position 0 0 0)))
;; plist
(define start (gvector 1 1 1))
;; start
(define end (gvector 1 1 1))
;; end
(define testcur (edge:spline plist start end))
;; testcur
; Create a curve law.
(define curlaw
    (law "map(term(cur(edge1),1),edge1)"
    testcur))
;; curlaw
; Find its minimum.
(define min (law:nmin curlaw 0 1))
;; min
; Plot and mark that point.
(define minpoint (dl–item:point
    (curve:eval–pos testcur min)))
;; minpoint
(dl–item:display minpoint)
;; ()
; OUTPUT Result
```

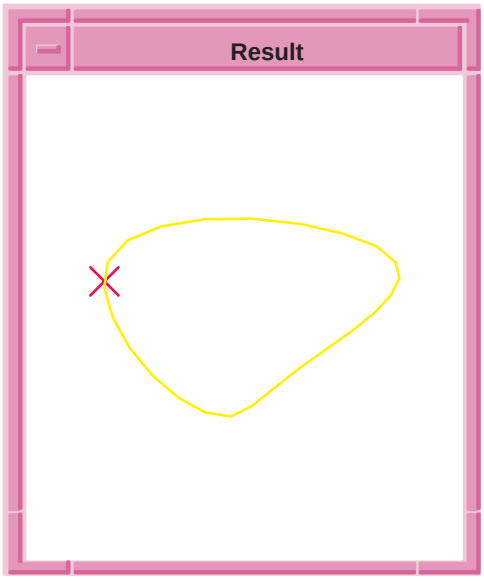

Figure 14-2. law:nmin – Minimum Point on a Test Curve

law:nroot

Scheme Extension: Laws, Analyzing Models

Action: Computes all of the roots of a law function over a given interval.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:nroot law low–lim up–lim)`

Arg Types:		
	law	law | string | real | position | par–pos | gvector
	low–lim	real
	up–lim	real

Returns: (real | (real ...))

Errors: None

Description: Returns a list of numbers that represent the roots of the law over a given range.

law can be a defined law, a law string, or a constant number.

low–lim and up–lim are two real values that specify the interval.

Limitations: None

Example:
```
; law:nroot
; Find the roots of a sine wave.
(law:nroot "sin(x)" 0 10)
;; (0 3.14159265358979 6.2831853071295
    9.42477796085266)
```

law:nsolve

Scheme Extension: Laws, Analyzing Models

Action: Computes all the locations where two laws are equal to one another over a given interval.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:nsolve** law1 law2 low–lim up–lim)

Arg Types:

law1	law | string | real | position | par–pos | gvector
law2	law | string | real | position | par–pos | gvector
low–lim	real
up–lim	real

Returns: (real | (real ...))

Errors: None

Description: This returns a list of number of real value represents where the two laws are equal over a given range.

law1 can be a defined law, a law string, or a constant number.

law2 can be a defined law, a law string, or a constant number.

low–lim and up–lim are two real values that specify the interval.

Limitations: None

Example:
```
; law:nsolve
; Define two laws and find where equal on interval.
(law:nsolve "x" "x^3" -2 2)
;; (-1.0 0 1.0)
```

law:quintic

Scheme Extension: Laws, Mathematics

Action: Creates a quintic law given {a,b,f(a),f(b),f'(a),f'(b) f''(a) f''(b)}.

Filename: kern/kern_scm/law_scm.cxx

APIs: api_make_quintic

Syntax: `(law:quintic a b f–a f–b df–a df–b ddf–a ddf–b)`

Arg Types:

a	real
b	real
f–a	real
f–b	real
df–a	real
df–b	real
ddf–a	real
ddf–b	real

Returns: boolean

Errors: None

Description: Produces a quintic polynomial with given boundary conditions for it, its first derivative, and its second derivative.

a and b specifies the boundary values.

f–a and f–b are the desired output of the law at a and b.

df–a and df–b are the desired output of the first derivative at a and b.

ddf–a and ddf–b are the desired output of the second derivative at a and b.

Limitations: None

Example:

```
; law:quintic
; Create a law.
(define law1 (law:quintic 0 1 2 3 4 5 6 7))
;; law1
(law:eval law1 0)
;; 2
(law:eval law1 1)
;; 3
(define dlaw (law:derivative law1))
;; dlaw
(define ddlaw (law:derivative dlaw))
;; ddlaw
(law:eval dlaw 0)
;; 3.99999999999998
(law:eval dlaw 1)
;; 5.00000000000003
(law:eval ddlaw 0)
;; 6
(law:eval ddlaw 1)
;; 7
```

law:reset–deriv

Scheme Extension: Laws, Analyzing Models

Action: Sets the derivative of i*th* driving variable.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:reset–deriv** a–law i)

Arg Types:

a–law	law | string | real | position | par–pos | gvector
i	integer

Returns: unspecified

Errors: None

Description: Refer to action.

a–law can be a defined law, a law string, or a constant number.

i specifies the order of derivative.

Limitations: None

Example:
```
; law:reset–deriv
; Create a law
(define a–law (law ″x^2+3″))
;; a–law
; Create a deriv–law
(define deriv–law (law ″2*x″))
;; deriv–law
; Set the derivative
(law:set–deriv a–law 0 deriv–law)
;; #t
; Reset the derivative
(law:reset–deriv a–law 0)
;; #t
```

law:set–deriv

Scheme Extension: Laws, Analyzing Models

Action: Manually sets the derivative of i*th* driving variable.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:set–deriv** a–law i deriv–law)

Arg Types:

a–law	law | string | real | position | par–pos | gvector
i	integer
deriv–law	law | string | real | position | par–pos | gvector

Returns: unspecified

Errors: None

Description: Refer to action.

a–law can be a defined law, a law string, or a constant number.

i specifies the order of derivative.

deriv–law specifies the closed form derivative of law:set–deriv.

Limitations: None

Example:

```
; law:set-deriv
; Create a law
(define a-law (law "x^2+3"))
;; a-law
; Create a deriv-law
(define deriv-law (law "2*x"))
;; deriv-law
; Set the derivative
(law:set-deriv a-law 0 deriv-law)
;; #t
```

law:simplify

Scheme Extension: Laws, Analyzing Models

Action: Creates a law which is the algebraic simplification of the given law.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law:simplify** law)

Arg Types:

law	law | string | real | position | par–pos | gvector

Returns: law

Errors: None

Description: The simplifier should always produce a law that is algebraically equivalent to the given law, but it may fail to simplify the law completely. Its original intent was to remove 0's and 1's from derivatives.

law can be a defined law, a law string, or a constant number.

Limitations: None

Example:

```
; law:simplify
; Create a new law.
(define law1 (law "x^2/x-3*x"))
;; law1
; Simplify the given law.
(define simplaw2 (law:simplify law1))
;; simplaw2
; Simplify the given law.
(define simplaw (law:simplify "x^2/x-3*x"))
;; simplaw
```

law:start

Scheme Extension: Laws

Action: Finds the start of the domain of a law.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: `(law:start in-law [term])`

Arg Types:

in–law	law
term	integer

Returns: boolean

Errors: None

Description: Returns the start of the domain of a law.

law specifies a law.

The optional term argument could be used to localize which term in the law is used.

Limitations: None

Example:

```
; law:start
; Define a profile to sweep for interesting shape
(define profile1 (wire-body:points
    (list (position 5 0 0)
    (position 10 0 0)
    (position 10 5 0)
    (position 5 5 0)
    (position 5 0 0))))
;; profile1
(define path1(edge:helix
    (position 0 0 0)
    (position 0 40 0)
    (gvector 1 0 0) 10 20 #t))
;; path1
(define helix1(sweep:law profile1 path1))
;; helix1
(define edge-list(entity:edges helix1))
;; edge-list
(define law1
    (law "cur(EDGE1)"
    (list-ref edge-list 3)))
;; law1
(law:start law1)
;; 0
(law:end law1)
;; 12.5663706143592
(define law2 (law "cur(EDGE1)"
    (list-ref edge-list 4)))
;; law2
(law:start law2)
;; 0
(law:end law2)
;; 5
```

law?

Scheme Extension: Laws

Action: Determines whether or not a Scheme object is of the law Scheme data type.

Filename: kern/kern_scm/law_scm.cxx

APIs: None

Syntax: (**law?** object)

Arg Types: object — scheme–object

Returns: boolean

Errors: None

Description: This extension returns a #t if the given object is a law Scheme data type. Otherwise, it returns a #f.

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

Limitations: None

Example:

```
; law?
; Create a new law.
(define law1 (law "x+x^2-cos(x)"))
;; law1
; Test to see if item is a law Scheme data type.
(law? law1)
;; #t
; Evaluate the given law at 1.5 radians
(define answer (law:eval law1 1.5))
;; answer
; Test to see if item is a law Scheme data type.
(law? answer)
;; #t
; This is true, because real numbers can be laws.
```

loop:find

Scheme Extension: Model Topology

Action: Gets loops shared by two entities.

Filename: kern/kern_scm/loop_scm.cxx

APIs: None

Syntax: (**loop:find** entity1 entity2)

Arg Types: entity1 — entity
entity2 — entity

Returns: loop

Errors: None

Description: This extension returns #f if no loop is found that is shared by both entities.

entity1 specifies the first entity.

entity2 specifies the second entity.

Limitations: None

Example:

```
; loop:find
; Create a solid block.
(define block1 (solid:block (position 20 30 0)
    (position 0 0 40)))
;; block1
; Create a solid cylinder.
(define cyl1 (solid:cylinder (position 0 0 0)
    (position –20 –20 –20) 15 3 (position –5 –5 0)))
;; cyl1
; Determine if any loops are shared.
(loop:find block1 cyl1)
;; #f
; Create an edge list.
(define list1 (entity:edges block1))
;; list1
(define one–edge (car list1))
;; one–edge
(define two–edge (car (cdr list1)))
;; two–edge
; Determine if any loops are shared.
(loop:find one–edge two–edge)
;; #[entity 16 1]
```

loop:type

Scheme Extension: Debugging

Action: Returns the type of given loop.

Filename: kern/kern_scm/loop_scm.cxx

APIs: None

Syntax: (**loop:type** loop)

Arg Types:	loop	entity

Returns: string

Errors: None

Description: This returns a string specifying the type of loop. This includes “Periphery loop”, “Hole”, “Separation loop”, “Separation loop (U)”, “Separation loop (V)”, “Separation loop (UV)”, and “Unknown type of loop”

loop is an entity representing a loop.

Limitations: None

Example:

```
; loop:type
; Create a model with loops
(define profile1 (edge:linear
    (position 5 0 0) (position 10 0 0)))
;; profile1
(is:helix profile1)
;; #f
(define path1(edge:helix
    (position 0 0 0)
    (position 0 40 0)
    (gvector 1 0 0) 10 20 #t))
;; path1
(define helix1(sweep:law profile1 path1))
;; helix1
(define loop–list (entity:loops helix1))
;; loop–list
(loop:type (list–ref loop–list 0))
;; ”Periphery loop”
```

loop?

Scheme Extension: Model Topology

Action: Determines if a Scheme object is a loop.

Filename: kern/kern_scm/ent_scm.cxx

APIs: None

Syntax: (**loop?** object)

Arg Types:	object	scheme–object

Returns: boolean

Errors: None

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

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

Limitations: None

Example:

```
; loop?
; Create a solid cylinder.
(define cyl1 (solid:cylinder (position 0 0 0)
    (position 8 8 8) 32))
;; cyl1
; Find the loops of the cylinder.
(define loops1 (entity:loops cyl1))
;; loops1
; Determine if the cylinder's loop is
; actually a loop.
(loop? (car (cdr loops1)))
;; #t
(loop? cyl1)
;; #f
```

lump?

Scheme Extension: Model Topology

Action: Determines if a Scheme object is a lump.

Filename: kern/kern_scm/ent_scm.cxx

APIs: None

Syntax: (**lump?** object)

Arg Types:	object	scheme–object

Returns: boolean

Errors: None

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

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

Limitations: None

Example:

```
; lump?
; Create a solid cylinder.
(define cyl1 (solid:cylinder (position 0 0 0)
    (position 8 8 8) 32))
;; cyl1
; Find the lumps of the cylinder.
(define limps1 (entity:lumps cyl1))
;; limps1
; Determine if the cylinder's lump is
; actually a lump.
(lump? (car limps1))
;; #t
(lump? cyl1)
;; #f
```

mixed–body?

Scheme Extension: Model Topology

Action: Determines if a Scheme object is a mixed body.

Filename: kern/kern_scm/ent_scm.cxx

APIs: None

Syntax: (**mixed–body?** object)

Arg Types: object scheme–object

Returns: boolean

Errors: None

Description: A body is considered to be mixed if it has at least one face and at least one LUMP with a SHELL, and at least one WIRE and one FACE are attached to that SHELL, but no WIREs are attached to the BODY directly.

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

Limitations: None

Example:

```
; mixed–body?
; Create a solid block.
(define block1
    (solid:block (position –20 –20 –20)
    (position 20 20 20)))
;; block1
; Determine if the block is a mixed body.
(mixed–body? block1)
;; #f
; Create a wire body.
(define wirebody1
    (wire–body:points (list (position 20 0 0)
    (position 50 0 0))))
;; wirebody1
; Determine if the wire body is a mixed body.
(mixed–body? wirebody1)
;; #f
; Unite the entities.
(define mixed
    (bool:nonreg–unite block1 wirebody1))
;; mixed
; Determine if the resulting body is a mixed body.
(mixed–body? mixed)
;; #t
```

monitor:file

Scheme Extension: Model Topology

Action: Opens or closes a monitor file, which records the text in the I/O window.

Filename: kern/kern_scm/dbg_scm.cxx

APIs: None

Syntax: (**monitor:file** filename)

Arg Types:	filename	string | boolean

Returns: string | boolean

Errors: None

Description: The monitor file records all text going to the Scheme I/O window in a file.

If filename is a string, a file of that name is opened and recording of the output begins. If filename is #f, the file is closed.

Limitations: None

Example:
```
; monitor:file
; Open a monitor file
(monitor:file ”record1”)
;; ”record1”
```

option:get

Scheme Extension: Modeler Control

Action: Gets the value of an option.

Filename: kern/kern_scm/opt_scm.cxx

APIs: None

Syntax: (**option:get** option–string)

Arg Types: option–string string

Returns: integer | real | boolean | position | string

Errors: None

Description: The command option:list displays the available options.

The argument option–string is generally a quoted text. When entering an option, the text can be abbreviated to the portion preceding the pound (#) sign. For example, brief_s#urface_debug can be abbreviated as brief_s.

Limitations: None

Example:

```
; option:get
; List the options available.
(option:list)
;; (("abl_c#aps" . #t)
;; ("abl_off_x#curves" . #f)
;; ("abl_rem#ote_ints" . #f)
;; ("abl_require_on#_support" . #t)
;; ("abort_on_illegal_surface" . #t)
;; ("adaptive#_grid" . #f)
;; ("adaptive_t#riangles" . #f)
;; ("add_bl_atts" . #f)
;; ("addr#ess_debug" . #t)
;; ("align_corners" . #t)
;; ("align_first_wire" . #f)
;; ("all_free_edges" . #f)
;; ("angular_control" . #t)
;; ("anno#tations" . #f)
;; ("api_checking" . #t)
;; ("approx#_eval" . #t)
;; ("approx-vbl_off#set" . #t)
;; ("auto_disp#lay" . #t)
;; ("backup_boxes" . #t)
;; ("bb_immediate_close" . #f)
;; ("bend_debug_file" . "")
;; ("bend_self_int" . #f)
;; ("bhl_smooth_edges" . #t)
;; ("bhl_smooth_surfaces" . #t)
;; ("bhl-use_ds_patch" . #f)
;; ("bhl-use_iso_approx" . #f)
;; ("bhl-use_iso_patch" . #f)
;; ("binary_format" . #f)
; .
; .
; .
; Get the value of an option.
(option:get "addr")
;; #t
```

option:list

Scheme Extension: Modeler Control

Action: Gets a list of available options and their associated values.

Filename: kern/kern_scm/opt_scm.cxx

APIs: None

Syntax: (**option:list**)

Arg Types: None

Returns: ((string . integer | real | boolean | position | string) ...)

Errors: None

Description: The extension returns a list of pairs of option names and values. The list displayed is bound by the variable print–length. It may be insufficient to display the whole list. If the whole list is not printed, try increasing the print length to 200 using the following:

```
(fluid–let ((print–length 200))
    (print (option:list)))
```

Limitations: None

Example:

```
; option:list
; Get a list of some of the options and their values.
; If not all of the options are displayed using
; the list command try the following command
; which increases the print-length
; (fluid-let ((print-length 200))
;   (print (option:list)))
(option:list)
;; (("abl_c#aps" . #t)
;; ("abl_off_x#curves" . #f)
;; ("abl_rem#ote_ints" . #f)
;; ("abl_require_on#_support" . #t)
;; ("abort_on_illegal_surface" . #t)
;; ("adaptive#_grid" . #f)
;; ("adaptive_t#riangles" . #f)
;; ("add_bl_atts" . #f)
;; ("addr#ess_debug" . #t)
;; ("align_corners" . #t)
;; ("align_first_wire" . #f)
;; ("all_free_edges" . #f)
;; ("angular_control" . #t)
;; ("anno#tations" . #f)
;; ("api_checking" . #t)
;; ("approx#_eval" . #t)
;; ("approx-vbl_off#set" . #t)
;; ("auto_disp#lay" . #t)
;; ("backup_boxes" . #t)
;; ("bb_immediate_close" . #f)
;; ("bend_debug_file" . "")
;; ("bend_self_int" . #f)
;; ("bhl_smooth_edges" . #t)
;; ("bhl_smooth_surfaces" . #t)
;; ("bhl-use_ds_patch" . #f)
;; ("bhl-use_iso_approx" . #f)
;; ("bhl-use_iso_patch" . #f)
;; ("binary_format" . #f)
; .
; .
; .
```

option:reset

Scheme Extension: Modeler Control

Action: Sets the list of available options to their initial values.

Filename: kern/kern_scm/opt_scm.cxx

APIs: None

Syntax: (**option:reset**)

Arg Types: None

Returns: boolean

Errors: None

Description: The default values for the options are presented in the option templates.

Limitations: None

Example:

```
; option:reset
; List the options available.
(option:list)
;; ;; ((”abl_c#aps” . #t)
;; (”abl_off_x#curves” . #f)
;; (”abl_rem#ote_ints” . #f)
;; (”abl_require_on#_support” . #t)
;; (”abort_on_illegal_surface” . #t)
;; (”adaptive#_grid” . #f)
;; (”adaptive_t#riangles” . #f)
;; (”add_bl_atts” . #f)
;; (”addr#ess_debug” . #t)
;; (”align_corners” . #t)
;; (”align_first_wire” . #f)
;; (”all_free_edges” . #f)
;; (”angular_control” . #t)
;; (”anno#tations” . #f)
;; (”api_checking” . #t)
;; (”approx#_eval” . #t)
;; (”approx-vbl_off#set” . #t)
;; (”auto_disp#lay” . #t)
;; (”backup_boxes” . #t)
;; (”bb_immediate_close” . #f)
;; (”bend_debug_file” . ””)
;; (”bend_self_int” . #f)
;; (”bhl_smooth_edges” . #t)
;; (”bhl_smooth_surfaces” . #t)
;; (”bhl-use_ds_patch” . #f)
;; (”bhl-use_iso_approx” . #f)
;; (”bhl-use_iso_patch” . #f)
;; (”binary_format” . #f)
;; (”bl_ana#lytic_spine” . #t)
; .
; .
; .
; Set the value of an option.
(option:set ”sphere_silhouettes” ”on”)
;; #t
; Reset the option to its previous value.
(option:reset)
;; #t
```

option:set

Scheme Extension: Modeler Control

Action: Sets the value of an option.

Filename: kern/kern_scm/opt_scm.cxx

APIs: None

Syntax: `(option:set option–string value)`

Arg Types:

option–string	string
value	real | integer | boolean | position | string

Returns: real | integer | boolean | position | string

Errors: None

Description: The return value is the previous option value. If option–string is not a valid option, an error message returns.

Refer to the option reference manuals for information on each specific options. Availability of the options is based on the products that have been included in your executable.

The argument option–string specifies the type of option to set.

value specifies the search value or attribute (boolean, integer, real, string, position) associated with the option–string.

Limitations: None

Example:

```
; option:set
; List the options available.
(option:list)
;; ;; ((”abl_c#aps” . #t)
;; (”abl_off_x#curves” . #f)
;; (”abl_rem#ote_ints” . #f)
;; (”abl_require_on#_support” . #t)
;; (”abort_on_illegal_surface” . #t)
;; (”adaptive#_grid” . #f)
;; (”adaptive_t#riangles” . #f)
;; (”add_bl_atts” . #f)
;; (”addr#ess_debug” . #t)
;; (”align_corners” . #t)
;; (”align_first_wire” . #f)
;; (”all_free_edges” . #f)
;; (”angular_control” . #t)
;; (”anno#tations” . #f)
;; (”api_checking” . #t)
;; (”approx#_eval” . #t)
;; (”approx-vbl_off#set” . #t)
;; (”auto_disp#lay” . #t)
;; (”avoid_box_backups” . #f)
;; (”backup_boxes” . #t)
;; (”bb_immediate_close” . #f)
;; (”bend_debug_file” . ””)
;; (”bend_self_int” . #f)
;; (”bhl_smooth_edges” . #t)
;; (”bhl_smooth_surfaces” . #t)
;; (”bhl-use_ds_patch” . #f)
;; (”bhl-use_iso_approx” . #f)
;; (”bhl-use_iso_patch” . #f)
;; (”bin#ary_format” . #f)
;; (”bl_ana#lytic_spine” . #t)
; .
; .
; .
; Set the value of an option.
(option:set ”sphere_silhouettes” ”on”)
;; #t
```

par–pos

Scheme Extension: Mathematics

Action: Creates a new *uv* parametric position given coordinate values *u* and *v*.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos u v)`

Arg Types:

u	real
v	real

Returns: par–pos

Errors: None

Description: Builds and returns a par–pos object for parametric location *uv*.

u defines the *u* coordinate in parameter space.

v defines the *v* coordinate in parameter space.

Limitations: None

Example:

```
; par-pos
; Create a par-pos object.
(par-pos 3 3)
;; #[par-pos 3 3]
```

par–pos:copy

Scheme Extension: Mathematics

Action: Creates a new par–pos by copying an existing par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos:copy par-pos)`

Arg Types:

par–pos	par–pos

Returns: par–pos

Errors: None

Description: Refer to Action.

par–pos specifies a par–pos object to be copied.

Limitations: None

Example:

```
; par-pos:copy
; Create a par-pos object.
(define par1 (par-pos 3 3))
;; par1
; Create a new par-pos by copying an
; existing par-pos.
(par-pos:copy par1)
;; #[par-pos 3 3]
```

par–pos:distance

Scheme Extension: Mathematics, Analyzing Models

Action: Gets the 2D distance between two par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: (**par-pos:distance** par-pos1 par-pos2)

Arg Types:

par–pos1	par–pos
par–pos2	par–pos

Returns: real

Errors: None

Description: Refer to Action.

par–pos1 defines the start location.

par–pos2 defines the end location.

Limitations: None

Example:

```
; par-pos:distance
; Determine the distance between two par-pos.
(par-pos:distance (par-pos 0 1) (par-pos 0 20))
;; 19
```

par–pos:set!

Scheme Extension: Mathematics

Action: Sets the *u* and *v*–components of a par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos:set! par-pos {u v})`

Arg Types:

par–pos	par–pos
u	real
v	real

Returns: par–pos

Errors: None

Description: Refer to Action.

par–pos specifies the *u* and *v* values.

u specifies the value to replace the original u value specified in par–pos.

v specifies the value to replace the original v value specified in par–pos.

Limitations: None

Example:

```
; par-pos:set!
; Set new u- and v-coordinates for an
; existing par-pos.
(define par1 (par-pos 0 0))
;; par1
(par-pos:set! par1 3 5)
;; #[par-pos 3 5]
```

par–pos:set–u!

Scheme Extension: Mathematics

Action: Sets the *u*–component of a par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos:set-u! par-pos u)`

Arg Types:

par–pos	par–pos
u	real

Returns: real

Errors: None

Description: The *u*–coordinate is returned as a real.

par–pos identifies the original *u* and *v* values.

u specifies the value to replace the original *u* value specified in par–pos.

Limitations: None

Example:
```
; par–pos:set–u!
; Create a par–pos object.
(define par1 (par–pos 0 0))
;; par1
; Set a new u–coordinate for an existing par–pos.
(par–pos:set–u! par1 5)
;; 5
```

par–pos:set–v!

Scheme Extension: Mathematics

Action: Sets the *v*–component of a par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: (**par–pos:set–v!** par–pos v)

Arg Types:

par–pos	par–pos
v	real

Returns: real

Errors: None

Description: The *v*–coordinate is returned as a real.

par–pos identifies the original *u* and *v* values.

v specifies the value to replace the original *v* value specified in par–pos.

Limitations: None

Example:
```
; par–pos:set–v!
; Create a par–pos object.
(define par1 (par–pos 0 0))
;; par1
; Set a new v–coordinate for an existing par–pos.
(par–pos:set–v! par1 8)
;; 8
```

par–pos:u

Scheme Extension: Mathematics

Action: Gets the *u*–component of a par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos:u par-pos)`

Arg Types: par–pos par–pos

Returns: real

Errors: None

Description: Refer to Action.

par–pos identifies the *u* and *v* values.

Limitations: None

Example:

```
; par-pos:u
; Create a par-pos object.
(define par1 (par-pos 3 7))
;; par1
; Get a par-pos's u-coordinate.
(par-pos:u par1)
;; 3
```

par–pos:v

Scheme Extension: Mathematics

Action: Gets the *v*–component of a par–pos.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: `(par-pos:v par-pos)`

Arg Types: par–pos par–pos

Returns: real

Errors: None

Description: Refer to Action.

par–pos identifies the *u* and *v* values.

Limitations: None

Example:
```
; par–pos:v
; Create a par–pos object.
(define par1 (par–pos 3 7))
;; par1
; Get a par–pos's v–coordinate
(par–pos:v par1)
;; 7
```

par–pos?

Scheme Extension: Mathematics

Action: Determines if a Scheme object is a part.

Filename: kern/kern_scm/par_scm.cxx

APIs: None

Syntax: (**par–pos?** object)

Arg Types:	object	scheme–object

Returns: boolean

Errors: None

Description: Returns #t for par–pos objects, otherwise returns #f.

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

Limitations: None

Example:
```
; par–pos?
; Create a par–pos object.
(define par1 (par–pos 3 7))
;; par1
; Determine if an object is a par–pos.
(par–pos? par1)
;; #t
(par–pos? (gvector 0 0 0))
;; #f
(par–pos? –4)
;; #f
```

pattern

Scheme Extension: Patterns

Action: Constructs a pattern Scheme data type.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax:
```
(pattern {trans–vec [x–vec [y–vec [scale [z–vec
    [keep [root–transf]]]]]]} | in–pat |
    {positions [root]})
```

Arg Types:

trans–vec	law
x–vec	law
y–vec	law
scale	law
z–vec	law
keep	boolean
root–transf	transform
in–pat	pattern
positions	position | (position ...)
root	position

Returns: pattern

Errors: None

Description: This extension constructs a pattern based upon the supplied arguments. The pattern may be based upon laws or upon a list of positions. (The latter type may only be used to generate translational patterns. For more general kinds of list–based patterns, use the pattern:from–list extension). In the case of law–based patterns, the user need only provide laws necessary for the creation of the pattern.

trans–vec argument governs the translations.

x–vec, y–vec and z–vec specify the orientation of the body axes of the pattern elements. Together, they govern the rotations. If z–vec is omitted, the z axis is assumed to be given by the cross product of the x–vec and y–vec. It is therefore necessary to specify z–vec only if the pattern contains reflections.

scale defines a scale law to be applied to the existing pattern. The scale law may evaluate either to a scalar for uniform scaling or a gvector for non–uniform scaling.

keep may be specified as false (#f) to suppress the pattern elements whose indices lie within the ranges of the other laws.

root–transf may be used if the seed pattern element should be transformed. Without it, the seed remains unchanged.

in–pat specifies a pattern. If in–pat alone is supplied, the extension merely makes a copy of this.

positions may be used to provide a list of pattern positions.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

Limitations: None

Example:

```
; pattern
; Create a translation law and attach a domain.
(define trans–vec (law "DOMAIN
    (VEC (10*X, 5*Y, 0), 0, 5, 0, 4)"))
;; trans–vec
; Create orientation laws.
(define x–vec (law "VEC (COS (X*PI/4),
    SIN (X*PI/4), 0)"))
;; x–vec
(define y–vec (law "VEC (0,0,1)"))
;; y–vec
; Create a pattern from the laws.
(define pat (pattern trans–vec x–vec y–vec))
;; pat
```

pattern:alternating–keep

Scheme Extension: Patterns

Action: Creates a new pattern by applying an alternating keep filter to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_alternating_keep_pattern

Syntax:

```
(pattern:alternating–keep pat keep1 keep2
    which–dim [merge=#t])
```

Arg Types:

pat	pattern
keep1	real
keep2	real
which–dim	integer
merge	boolean

Returns: pattern

Errors: None

Description: Applies an alternating keep filter to an existing pattern, merging with any existing filter or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

keep1 and keep2 are successive Boolean keep values.

which–dim specifies the dimension in which the filter is applied.

If merge is false (#f), the existing pattern is replaced.

Limitations: None

Example:

```
; pattern:alternating–keep
; 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 x–vec (gvector 4 0 0))
;; x–vec
(define num_x 8)
;; num_x
(define y–vec (gvector 0 2 0))
;; y–vec
(define num_y 10)
;; num_y
```

```
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; modify the pattern
(define keep1 #f)
;; keep1
(define keep2 #t)
;; keep2
(define dim 1)
;; dim
(set! pat (pattern:alternating–keep pat
    keep1 keep2 dim))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)”
;; x–vec:      ”null_law”
;; y–vec:      ”null_law”
;; z–vec:      ”null_law”
;; scale:      ”null_law”
;; keep:       ”null_law”
;; no list]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:alternating–scale

Scheme Extension: Patterns

Action: Creates a new pattern by applying an alternating scale to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_alternating_scale_pattern

Syntax: (**pattern:alternating–scale** pat scale1 scale2 which–dim root [merge=#t])

Arg Types:		
	pat	pattern
	scale1	real | gvector
	scale2	real | gvector
	which–dim	integer
	root	position
	merge	boolean

Returns: pattern

Errors: None

Description: Applies an alternating scale to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

scale1 and scale2 specifies the alternating scales. The scale values must be greater than zero. They may be given as vectors when non–uniform scaling is desired.

which–dim specifies the dimension in which the scale is applied.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

If merge is false (#f), the existing pattern is replaced.

Limitations: None

Example:

```
; pattern:alternating-scale
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 8)
;; num_x
```

```
(define y–vec (gvector 0 2 0))
;; y–vec
(define num_y 10)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; modify the pattern
(define scale1 0.8)
;; scale1
(define scale2 1.2)
;; scale2
(define dim 1)
;; dim
(define root origin)
;; root
(set! pat (pattern:alternating–scale
    pat scale1 scale2 dim root))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)”
;; x–vec:      ”null_law”
;; y–vec:      ”null_law”
;; z–vec:      ”null_law”
;; scale:      ”null_law”
;; keep:       ”null_law”
;; no list]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:check

Scheme Extension: Patterns

Action: Lists the elements in an entity that have a pattern attached.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_get_coedges, api_get_edges, api_get_faces, api_get_loops, api_get_lumps, api_get_shells, api_get–vertices, api_get_wires

Syntax: (**pattern:check** entity [type–list] [color])

Arg Types:		
	entity	entity
	type–list	string | (string ...)
	color	boolean

Returns: boolean

Errors: None

Description: Checks for the presence of patterns on the BODY, LUMPs, SHELLs, FACEs, and LOOPs of an entity and generates a list of the elements with patterns. Returns #t when elements with patterns exist and #f when there are no elements with patterns.

entity specifies an entity.

Defining type–list (’bodies, ’lumps, ’shells, ’faces, or ’loops) generates a more selective check. By default, the function checks all entities.

If color is specified as #t, the faces, edges, and vertices that are patterned are colored in the display.

Limitations: None

Example:

```
; pattern:check
; Create a sphere at the origin to demonstrate
;   command.
(define s (solid:sphere (position 0 0 0) 10))
;; s
; Define a 2x2 hex pattern.
(define p (pattern:hex (gvector 0 1 1)
    (gvector 40 0 0) 2 2))
;; p
; Create an entity using sphere in the pattern.
(define hexpat (entity:pattern s p))
;; hexpat
; Check that hexpat is a pattern.
(pattern:check hexpat)
; entities with patterns:
;          1 bodies
;          4 lumps
;          4 shells
;          0 wires
;          4 faces
;          4 surfaces
;          1 loops
;          0 coedges
;          0 pcurves
;          0 edges
;          0 curves
;          0 vertices
;          0 points
;          All entities examined have patterns
;; #t
; Check faces only.
(pattern:check hexpat 'faces)
;       face: (entity 11 1)
;       face: (entity 12 1)
;       face: (entity 13 1)
;       face: (entity 14 1)
; entities with patterns:
;       4 faces
; All entities examined have patterns
;; #t
```

pattern:circular?

Scheme Extension: Patterns

Action: Determines whether or not a pattern lies on a given circle.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: `(pattern:circular? pat {face | {center axis}})`

Arg Types:

pat	pattern
face	face
center	position
axis	gvector

Returns: boolean

Errors: None

Description: This extension is limited in that the pattern must be defined.

This extension returns #t if a pattern lies on the circular surface associated with the face or, alternately, with the combination of center and axis. Otherwise, it returns #f.

pat specifies a pattern.

face specifies a circular face.

center and axis together specify a circular surface.

Limitations: None

Example:

```
; pattern:circular?
; ; choose an axis and a center defining a circle
(define center (position 5 6 3))
;; center
(define normal (gvector 0 0 1))
;; normal
; make a pattern
(define pat (pattern:cylindrical
    center normal 4 10 3))
;; pat
; does the result have circular symmetry about center
; and normal?
(pattern:circular? pat center normal)
;; #f
; what about this pattern?
(set! pat (pattern:elliptical center normal 5))
;; trans-vec:  "DOMAIN(VEC(5,6,3)+VEC(0,0,1)
;; *3*X2,0,3,0,9)"
;; x-vec:      "VEC(-1,0,0)*COS
;; (1.570796326794897*X+0)
;;  +VEC(0,1,0)*SIN(1.570796326794897*X+0)"
;; y-vec:      "VEC(0,0,1)"
;; z-vec:      "null_law"
;; scale:      "null_law"
;; keep:       "null_law"
;; no list]
(pattern:circular? pat center normal)
;; #t
```

pattern:compose

Scheme Extension: Patterns

Action: Creates a new pattern by composing two existing ones.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:compose** pat1 pat2)

Arg Types:		
	pat1	pattern
	pat2	pattern

Returns: pattern

Errors: None

Description: Refer to Action.

pat1 and pat2 specifies the pattern.

Limitations: None

Example:

```
; pattern:compose
; 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 x-vec (gvector 2 0 0))
;; x-vec
(define num_x 4)
;; num_x
(define y-vec (gvector 1 3 0))
;; y-vec
(define num_y 5)
;; num_y
(define pat1
    (pattern:linear x-vec num_x y-vec num_y))
;; pat1
; make another pattern
(set! x-vec (gvector 20 0 0))
;; #[gvector 2 0 0]
(set! num_x 3)
;; 4
(set! y-vec (gvector 10 30 0))
;; #[gvector 1 3 0]
(set! num_y 2)
```

```
;; 5
(define pat2
    (pattern:linear x-vec num_x y-vec num_y))
;; pat2
; compose the two patterns
(define pat (pattern:compose pat1 pat2))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
 ;; body
```

pattern:concatenate

Scheme Extension: Patterns

Action: Creates a new pattern by concatenating two existing ones.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:concatenate** pat1 pat2 [transform])

Arg Types:		
	pat1	pattern
	pat2	pattern
	transform	transform

Returns: pattern

Errors: None

Description: Creates a new pattern by concatenating two existing ones, optionally applying a transform to the second.

pat1 specifies the first pattern.

pat2 specifies the second pattern.

transform is an optional argument specifying the transformation to be applied to the second pattern.

Limitations: None

Example:

```
; pattern:concatenate
; 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 x–vec (gvector 2 0 0))
;; x–vec
(define num_x 8)
;; num_x
(define pat1 (pattern:linear x–vec num_x))
;; pat1
; make another pattern
(define center (position –4 2 3))
;; center
(define normal (gvector 0 0 1))
;; normal
(define num 8)
;; num
(define pat2 (pattern:elliptical center normal num))
;; pat2
; concatenate the patterns
(define pat (pattern:concatenate pat1 pat2))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:coords–to–index

Scheme Extension: Patterns

Action: For the specified pattern, returns the pattern index associated with the specified pattern coordinates.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:coords–to–index** pat coords)

Arg Types:

pat	pattern
coords	pair

Returns: integer

Errors: None

Description: Refer to Action.

pat specifies a pattern.

coords specifies the pattern coordinates.

Limitations: None

Example:

```
; pattern:coords–to–index
; make a pattern
(define center (position 1 2 3))
;; 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
; find the pattern index of a specific element
(define coords (list 3 2))
;; coords
(define index (pattern:coords–to–index pat coords))
;; index
```

pattern:copy

Scheme Extension: Patterns

Action: Makes a copy of a pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:copy** pat)

Arg Types:	pat	pattern

Returns: pattern

Errors: None

Description: Refer to Action.

pat specifies a pattern.

Limitations: None

Example:

```
; pattern:copy
; make a pattern
(define center (position 1 2 3))
;; 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
; copy the pattern
(define pat_copy (pattern:copy pat))
;; pat_copy
```

pattern:cylindrical

Scheme Extension: Patterns

Action: Constructs a pattern with cylindrical symmetry.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_cylindrical_pattern

Syntax:
```
(pattern:cylindrical {center normal | face}
    num–angular [num–axial=0] [distance=0]
    [alternating=#f])
```

Arg Types:

center	position
normal	gvector
face	face
num–angular	integer
num–axial	integer
distance	real
alternating	boolean

Returns: pattern

Errors: None

Description: Refer to Action.

center and normal together specify the radius for the circular pattern.

face specifies a cylindrical face.

num–angular specifies the number of transverse elements in the pattern.

num–axial specifies the number of longitudinal elements in the pattern.

distance specifies the distance between the circular pattern layers.

If alternating is #t, adjacent layers are staggered in angle.

Limitations: None

Example:

```
; pattern:cylindrical
; Create a cylindrical pattern using defaults only.
(define s (solid:sphere (position –15 –10 0) 3))
;; s
(define p (pattern:cylindrical (position –10 –10 –10)
    (gvector 1 0 1) 5 10 5))
;; p
; Attach the defined pattern to the sphere.
(define attach (entity:pattern s p))
;; attach
; Create an alternating cylindrical pattern.
; Roll back 1 step and redefine the pattern.
(roll)
;; –1
(define p (pattern:cylindrical (position 0 0 0)
    (gvector 0 0 1) 15 5 10 #t))
;; p
; Attach the defined pattern to the sphere.
(define attach (entity:pattern s p))
;; attach
; Create an alternating cylindrical pattern.
; Roll back 1 step and redefine the pattern.
(roll)
;; –1
(define p (pattern:cylindrical (position 0 0 0)
    (gvector 1 1 0) 5 7 10))
;; p
; Attach the defined pattern to the sphere.
(define attach (entity:pattern s p))
;; attach
```

pattern:cylindrical?

Scheme Extension: Patterns

Action: Determines whether or not a pattern lies on a given cylinder.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:cylindrical?** pat {face | {center axis}})

Arg Types:		
	pat	pattern
	face	face
	center	position
	axis	gvector

Returns: boolean

Errors: None

Description: This extension returns #t if a pattern lies on the cylindrical surface. Otherwise, it returns #f.

pat specifies a pattern.

face specifies a cylindrical face.

center and axis together specify a cylindrical surface.

Limitations: None

Example:

```
; pattern:cylindrical?
; Choose an axis and a center defining a cylinder.
(define center (position 5 6 3))
;; center
(define normal (gvector 0 0 1))
;; normal
; Make a pattern.
(define pat (pattern:cylindrical
    center normal 4 10 3))
;; pat
; Does the result have cylindrical symmetry about
; the center and normal?
(pattern:cylindrical? pat center normal)
;; #t
; Try another pattern.
(set! pat (pattern:elliptical center normal 5))
;; #[pattern
;; trans-vec:  "DOMAIN(VEC(5,6,3)+VEC
;; (0,0,1)*3*X2,0,3,0,9)"
;; x-vec:      "VEC(-1,0,0)*COS
;; (1.570796326794897*X+0)
;; +VEC(0,1,0)*SIN
;; (1.570796326794897*X+0)"
;; y-vec:   "VEC(0,0,1)"
;; z-vec:   "null_law"
;; sale:    "null_law"
;; keep:    "null_law"
;; no list]
(pattern:cylindrical? pat center normal)
;; #t
; Try yet another pattern.
```

```
(set! pat (pattern:elliptical center
    (gvector 1 0 0) 5))
;; #[pattern
;; trans-vec:  "VEC(5,6,3)"
;; x-vec:      "DOMAIN(VEC(-1,0,0)
;; *COS(1.256637061435917*X)
;; +VEC(0,-1,0)*SIN
;;  (1.256637061435917*X),0,4)"
;; y-vec: "VEC(0,0,1)"
;; z-vec: "null_law"
;; scale: "null_law"
;; keep:  "null_law"
;; no list]
(pattern:cylindrical? pat center normal)
;; #f
```

pattern:edge

Scheme Extension: Patterns

Action: Creates a pattern parallel to an edge.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_edge_pattern

Syntax:
```
(pattern:edge entity-list number root
    [on-endpoints=#f] [normal-dir tangent-dir])
```

Arg Types:

entity–list	entity | (entity...)
number	integer
root	position
on–endpoints	boolean
normal–dir	gvector
tangent–dir	gvector

Returns: pattern

Errors: None

Description: Creates a one–dimensional pattern of number elements, equally spaced in parameter space, parallel to an edge.

entity–list argument may consist of an edge, a face and one of its bounding edges, or a coedge.

root specifies the position (which can be on or off the seed pattern entity, as desired) to be mapped to the pattern sites.

If on–endpoints is set to #t, the pattern could be extended to the endpoints of the edge.

normal–dir and tangent–dir specify the directions, relative to the seed entity, that are mapped to the normal and tangent directions of the edge, respectively. If normal–dir and tangent–dir are specified, the pattern members instead follow the curvature of the edge. By default, pattern members are oriented identically to one another. (When entity–list consists entirely of a single edge, a rail law is generated to determine the normal direction of the edge.)

Limitations: None

Example:

```
; pattern:edge
; 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 axis_start (position 0 –25 0))
;; axis_start
(define axis_end (position 0 25 0))
;; axis_end
(define start–dir (gvector 1 0 0))
;; start–dir
(define radius 5)
;; radius
(define thread_distance 10)
```

```
;; thread_distance
(define edge (edge:helix axis_start axis_end
    start–dir radius thread_distance))
;; edge
(define num 50)
;; num
(define root origin)
;; root
(define on–endpoints #f)
;; on–endpoints
(define rail–dir (gvector 1 0 0))
;; rail–dir
(define tangent–dir (gvector 0 1 0))
;; tangent–dir
(define pat (pattern:edge edge num
    root on–endpoints rail–dir tangent–dir))
;; pat
(entity:delete edge)
;; ()
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:elliptical

Scheme Extension: Patterns

Action: Creates an elliptical pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_elliptical_pattern

Syntax:

```
(pattern:elliptical center normal num
    [not–rotate=#f root [angle=360
    [ratio=1 major–axis]]])
```

Arg Types:		
	center	position
	normal	gvector
	num	integer
	not–rotate	boolean
	root	position
	angle	real
	ratio	real
	major–axis	gvector

Returns: pattern

Errors: None

Description: Creates a one–dimensional elliptical pattern defined by an axis of rotation.

center and normal define the (global) position and orientation of the axis.

num argument defines the number of entities in the pattern.

When not–rotate is #t, the elements of the pattern are kept in a fixed, relative orientation. In which case, root is required because it defines the position mapped to the pattern sites.

angle defines the angular extent of the pattern, with positive/negative values producing patterns proceeding clockwise/counter–clockwise (accordingly) about the normal vector.

ratio defines the ratio of minor/major radii of the pattern.

major–axis defines the direction of the ellipse's major axis.

Limitations: None

Example:

```
; pattern:elliptical
; Make a prism to demonstrate command.
(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
; Output Original
```

```
; Redefine position of origin.
(define origin (position 1 2 3))
;; origin
; Reposition prism.
(define transform (entity:transform prism
    (transform:axes origin (gvector 1 0 0)
    (gvector 0 1 0))))
;; transform
; Make a pattern.
(define center (position 11 2 3))
;; center
(define normal (gvector 0 0 1))
;; normal
(define num 24)
;; num
(define not–rotate #f)
;; not–rotate
(define root origin)
;; root
(define ang 360)
;; ang
(define ratio 0.5)
;; ratio
(define major–axis (gvector 2 1 0))
;; major–axis
(define pat (pattern:elliptical center normal num
    not–rotate root ang ratio major–axis))
;; pat
; Apply the pattern to the prism.
(define body (entity:pattern prism pat))
;; body
; Output Result
```

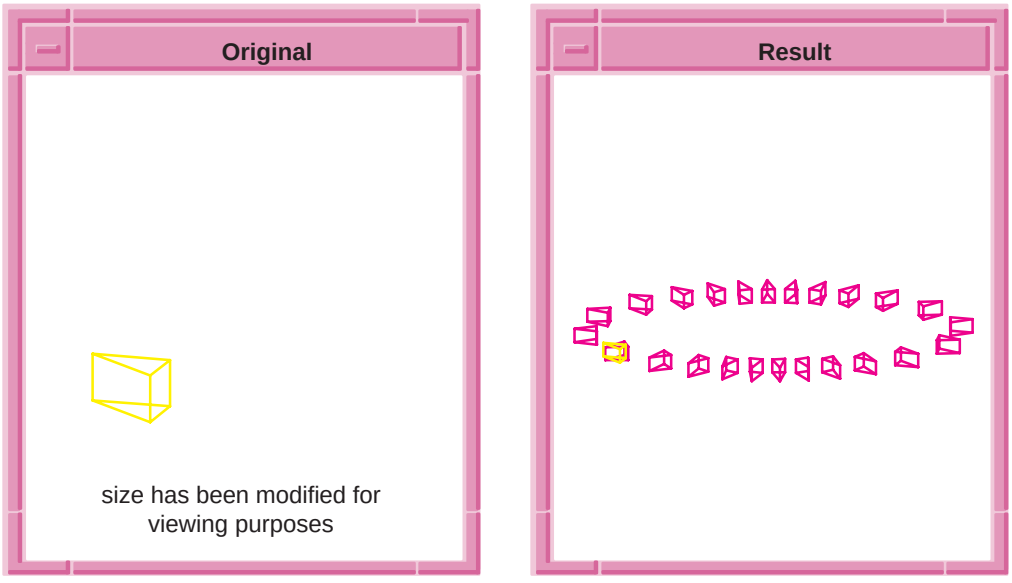

Figure 14-3. pattern:elliptical

pattern:from–list

Scheme Extension: Patterns

Action: Creates a pattern from a list of positions or transformations.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax:
```
(pattern:from–list {position–list [root]} |
    {transf–list [root–transf]})
```

Arg Types:

position–list	position | (position ...)
root	position
transf–list	transform | (transform ...)
root–transf	transform

Returns: pattern

Errors: None

Description: Creates a pattern from either the positions list or the transfs list. The optional arguments root and root–transf specify the transformation to the first pattern site.

Limitations: None

Example:

```
; pattern:from-list
; 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
(entity:set-color (point origin) RED)
;; ()
(define trans (entity:transform prism
    (transform:axes origin (gvector 1 0 0)
    (gvector 0 1 0))))
;; trans
; make a pattern
(define t0 (transform:axes (position 0 0 3)
    (gvector 1 0 0) (gvector 0 1 0)))
;; t0
(define t1 (transform:axes (position 0 6 3)
    (gvector 1 0 0) (gvector 0 1 0)))
;; t1
(define t2 (transform:axes (position 6 0 3)
    (gvector 1 0 0) (gvector 0 1 0)))
;; t2
(define t3 (transform:axes (position 6 6 3)
    (gvector 1 0 0) (gvector 0 1 0)))
;; t3
(define t4 (transform:axes (position 3 3 0)
    (gvector 0 1 0) (gvector 1 0 0)))
;; t4
(define t5 (transform:axes (position 0 0 -3)
    (gvector 0 1 0) (gvector 0 0 1)))
;; t5
(define t6 (transform:axes (position 0 6 -3)
    (gvector 0 1 0) (gvector 0 0 1)))
```

```
;; t6
(define t7 (transform:axes (position 6 0 –3)
    (gvector 0 1 0) (gvector 0 0 1)))
;; t7
(define t8 (transform:axes (position 6 6 –3)
    (gvector 0 1 0) (gvector 0 0 1)))
;; t8
(define root (transform:translation
    (gvector:from–to (position 0 0 0) origin)))
;; root
(define pat (pattern:from–list
    (list t0 t1 t2 t3 t4 t5 t6 t7 t8) root))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:get–transform

Scheme Extension: Patterns

Action: Returns the requested pattern transform.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:get–transform** pat [index1 [index2]])

Arg Types:		
	pat	pattern
	index1	integer
	index2	integer

Returns: transform

Errors: None

Description: This extension returns a transform associated with the pattern specified by pat. By default, the root transformation is returned.

pat specifies a pattern.

If index1 is given, the relative transformation for the pattern element indexed by its value is returned;

If both index1 and index2 are given, the transformation that generates the element indexed by index2 from the element indexed by index1 is returned.

Limitations: None

Example:

```
; pattern:get-transform
; make a pattern
(define axis_start (position 0 -25 0))
;; axis_start
(define axis_end (position 0 25 0))
;; axis_end
(define start-dir (gvector 1 0 0))
;; start-dir
(define radius 5)
;; radius
(define thread_distance 10)
;; thread_distance
(define edge (edge:helix axis_start axis_end
    start-dir radius thread_distance))
;; edge
(define num 50)
;; num
(define root (position 1 2 3))
;; root
(define on-endpoints #f)
;; on-endpoints
(define rail-dir (gvector 1 0 0))
;; rail-dir
(define tangent-dir (gvector 0 1 0))
;; tangent-dir
(define pat (pattern:edge edge num
    root on-endpoints rail-dir tangent-dir))
;; pat
(entity:delete edge)
;; ()
; get the root transform
(define t_root (pattern:get-transform pat))
;; t_root
;get the first transform
(define t_0 (pattern:get-transform pat 0))
;; t_0
; print the two transforms
(transform:print t_root)
; rotation no reflection no shear not identity
; translation part:
;  4.111957   -27.965300   -0.288970
; affine part:
;  0.951057    0.000000  -0.309017
```

```
; –0.294459   0.303314   –0.906253
;  0.093729   0.952891    0.288469
; scaling part:
; 1.000000
;; #[transform 118280584]
(transform:print t_0)
; rotation   no reflection   no shear   not identity
; translation part:
;  4.755283   –24.500000   –1.545085
; affine part:
; –0.294459    0.303314   –0.906253
;  0.951057    0.000000   –0.309017
; –0.093729   –0.952891   –0.288469
; scaling part:
; 1.000000
;; #[transform 118280584]
```

pattern:hex

Scheme Extension: Patterns

Action: Creates a hexagonal packing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_hex_pattern

Syntax: (**pattern:hex** normal xvec xnum ynum)

Arg Types:

normal	gvector
xvec	gvector
xnum	integer
ynum	integer

Returns: pattern

Errors: None

Description: Creates a hexagonal pattern in the plane defined by normal.

normal defines the plane.

xvec defines the x direction.

xnum and ynum defines the number of pattern entities in the x and y directions.

Limitations: None

Example:

```
; pattern:hex
; Create a prism to demonstrate command.
(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
; Enlarge prism for viewing.
(define zoom (zoom 4))
;; zoom
; OUTPUT Original

; Reposition 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
; Create a pattern.
(define normal (gvector 0 0 1))
;; normal
(define x-vec (gvector 2 0 0))
;; x-vec
(define num_x 6)
;; num_x
(define num_y 6)
;; num_y
(define pat (pattern:hex normal x-vec num_x num_y))
;; pat
; Apply the pattern to the prism.
(define body (entity:pattern prism pat))
;; body
; Adjust picture for better viewing.
(view:compute-extrema)
;; ()
; OUTPUT Result
```

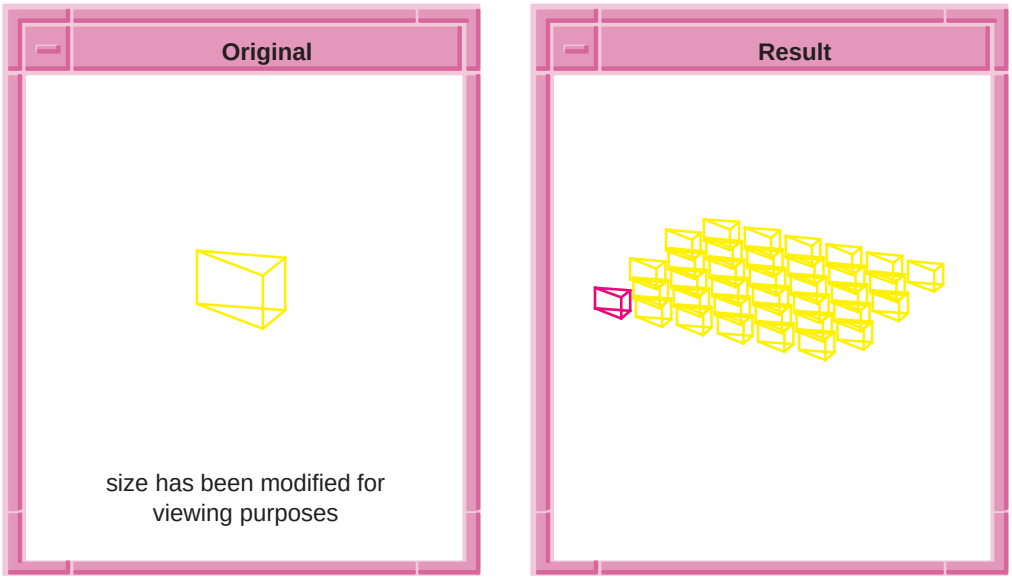

Figure 14-4. pattern:hex

pattern:hex–cylindrical

Scheme Extension: Patterns

Action: Creates a hexagonal pattern with cylindrical symmetry.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_hex_cylindrical_pattern

Syntax:
```
(pattern:hex-cylindrical center normal num-angular
    [num-axial=0] [radius=0])
```

Arg Types:

center	position
normal	gvector
num–angular	integer
num–axial	integer
radius	real

Returns: pattern

Errors: None

Description: Creates a two–dimensional hexagonal pattern with cylindrical symmetry.

center specifies the center position of the circular edge.

normal specifies the direction of extension.

num–angular specifies the number of angular elements in the pattern.

num–axial specifies the number of axial elements in the pattern.

radius specifies the distance between pattern elements.

Limitations: None

Example:

```
; pattern:hex-cylindrical
; 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
(entity:set-color (point origin) RED)
;; ()
(define transform (entity:transform prism
    (transform:axes origin
     (gvector 1 0 0) (gvector 0 1 0))))
;; transform
; make a pattern
(define center (position 6 2 3))
;; center
(define normal (gvector 0 1 0))
;; normal
(define num-angular 8)
;; num-angular
(define num-axial 6)
;; num-axial
(define spacing 3)
;; spacing
(define pat (pattern:hex-cylindrical
    center normal num-angular num-axial spacing))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:index–to–coords

Scheme Extension: Patterns

Action: For the specified pattern, returns the pattern coordinates associated with the specified pattern index.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: `(pattern:index–to–coords pat index)`

Arg Types:

pat	pattern
index	integer

Returns: (real ...)

Errors: None

Description: Refer to Action.

pat specifies a pattern.

index specifies a pattern index.

Limitations: None

Example:

```
; pattern:index-to-coords
; 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 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 pat1
    (pattern:radial center normal
    num-radial num-angular spacing))
;; pat1
; find the pattern coords of a specific element
(define index 13)
;; index
(define coords (pattern:index-to-coords pat1 index))
;; coords
```

pattern:keep

Scheme Extension: Patterns

Action: Creates a new pattern by applying a keep law to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:keep** pat keep [merge=#t])

Arg Types:

pat	pattern
keep	law
merge	boolean

Returns: pattern

Errors: None

Description: Applies a keep law to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies an exisiting pattern.

keep specifies a law.

If merge is specified as #f, the existing pattern is replaced. The default value is #t.

Limitations: None

Example:

```
; pattern:keep
; create a linear pattern in two dimensions
(define pat (pattern:linear
    (gvector 10 0 0) 5 (gvector 0 8 4) 4))
;; pat
; create a keep law that keeps only elements
; for which x + y < 5
(define keep (law "X + Y < 5"))
;; keep
; attach the keep law to the pattern
(set! pat (pattern:keep pat keep))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(10,0,0)+X2*VEC
;; (0,8,4),0,4,0,3)"
;; x-vec:  "null_law"
;; y-vec:  "null_law"
;; z-vec:  "null_law"
;; scale:  "null_law"
;; keep:   "null_law"
;; no list]
```

pattern:linear

Scheme Extension: Patterns

Action: Creates a linear pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_linear_pattern

Syntax: `(pattern:linear xvec xnum [yvec ynum [zvec znum]])`

Arg Types:

xvec	gvector
xnum	integer
yvec	gvector
ynum	integer
zvec	gvector
znum	integer

Returns: pattern

Errors: None

Description: Creates a linear pattern in one, two, or three dimensions based on the number of input arguments.

xvec, yvec and zvec specify the pattern directions (these do not have to be in the coordinate directions or orthogonal.)

xnum, ynum and znum specify the number of repetitions along xvec, yvec and zvec respectively.

Limitations: None

Example:

```
; pattern:linear
; create a sphere at the origin
(define s (solid:sphere (position 0 0 0)10))
;; s
; define the pattern in three dimensions with 3
;  repetitions along each axis
(define p (pattern:linear (gvector 30 10 0)
    3(gvector 0 30 0)3(gvector 0 0 30)3))
;; p
; create an entity using sphere in the pattern
(define spherepat (entity:pattern s p))
;; spherepat
(entity:check spherepat)
; checked:
;       27 lumps
;       27 shells
;       0 wires
;       27 faces
;       0 loops
;       0 coedges
;       0 edges
;       0 vertices
;; ()
```

pattern:linear–scale

Scheme Extension: Patterns

Action: Creates a new pattern by applying a linear scale to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_linear_scale_pattern

Syntax:

```
(pattern:linear–scale pat begin–scale end–scale
    which–dim root [merge=#t])
```

Arg Types:

pat	pattern
begin–scale	real | gvector
end–scale	real | gvector
which–dim	integer
root	position
merge	boolean

Returns: pattern

Errors: None

Description: Applies a linear scale, from begin–scale to end–scale, to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

begin–scale and end–scale specifies the starting and ending scales respectively. Both begin–scale and end–scale must be greater than zero. They may be given as vectors when non–uniform scaling is desired.

which–dim specifies the dimension in which the scale is applied.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

If merge is specifies as #f, the existing pattern is replaced. The default value is #t.

Limitations: None

Example:

```
; pattern:linear–scale
; 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 x–vec (gvector 4 0 0))
;; x–vec
(define num_x 8)
;; num_x
```

```
(define y–vec (gvector 0 2 0))
;; y–vec
(define num_y 10)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; modify the pattern
(define begin–scale 0.5)
;; begin–scale
(define end–scale 2.0)
;; end–scale
(define dim 0)
;; dim
(define root origin)
;; root
(set! pat (pattern:linear–scale
    pat begin–scale end–scale dim root))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)
;; x–vec:      ”null_law”
;; y–vec:      ”null_law”
;; z–vec:      ”null_law”
;; scale:      ”null_law”
;; keep:       ”null_law”
;; no list]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:make–entity

Scheme Extension: Patterns

Action: Creates a pattern entity data type composed of one pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_pattern_to_entity

Syntax: (**pattern:make–entity** pat)

Arg Types:	pat	pattern

Returns: pattern

Errors: None

Description: Creates a pattern entity which can be saved and restored from a SAT file. The pattern does not have to be attached to model geometry.

pat specifies a pattern.

Limitations: None

Example:

```
; pattern:make-entity
; make a pattern
(define normal (gvector  0 0 1))
;; normal
(define x-vec (gvector  2 0 0))
;; x-vec
(define num_x 6)
;; num_x
(define num_y 6)
;; num_y
(define pat (pattern:hex normal x-vec num_x num_y))
;; pat
; Make the associated APATTERN entity.
(define apat (pattern:make-entity pat))
;; apat
; Save myfile to a SAT file.
(part:save-selection apat "myfile.sat")
;; #t
```

pattern:mirror

Scheme Extension: Patterns

Action: Creates a new pattern that is a concatenation of the given pattern with its reflection through a specified plane.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:mirror** pat root normal)

Arg Types:

pat	pattern
root	position
normal	gvector

Returns: pattern

Errors: None

Description: Creates a pattern that is a concatenation of the pattern pat with its reflection.

pat specifies a pattern.

root specifies a point on the plane of reflection.

normal specifies the normal to the plane of reflection.

Limitations: None

Example:

```
; pattern:mirror
; 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 t1 (transform:translation (gvector 0 0 0)))
;; t1
(define t2 (transform:translation (gvector 2 2 0)))
;; t2
(define t3 (transform:translation (gvector 4 0 0)))
;; t3
(define t4 (transform:translation (gvector 0 4 0)))
;; t4
(define t5 (transform:translation (gvector 4 4 0)))
;; t5
(define pat (pattern:from–list
    (list t1 t2 t3 t4 t5)))
;; pat
; mirror the pattern about the origin
(define root (position 0 0 0))
;; root
(define normal (gvector 1 1 0))
;; normal
(set! pat (pattern:mirror pat root normal))
;; #[pattern [list]]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:move–element

Scheme Extension: Patterns

Action: Moves a single element within an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: `(pattern:move–element pat coords transf)`

Arg Types:

pat	pattern
coords	pair
transf	transform

Returns: pattern

Errors: None

Description: Moves the element specified by coords according to the transformation transf.

pat specifies a pattern.

coords specifies the element to be transformed.

transf specifies the transformation to be applied.

Limitations: None

Example:

```
; pattern:move-element
; 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
; Move the prism.
(define transform (entity:transform prism
    (transform:axes origin (gvector 1 0 0)
    (gvector 0 1 0))))
;; transform
; Make a pattern.
(define x-vec (gvector 2 0 0))
;; x-vec
(define num_x 8)
;; num_x
(define y-vec (gvector 0 3 0))
;; y-vec
(define num_y 8)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; OUTPUT Original
```

```
; Move one of the pattern elements.
(define disp (gvector 0.4 1.2 0.5))
;; disp
(define move (transform:translation disp))
;; move
(set! pat (pattern:move-element pat (list 3 2) move))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,3,0),0,7,0,7)"
;; x-vec:  "null_law"
;; y-vec:  "null_law"
;; z-vec:  "null_law"
;; scale:  "null_law"
;; keep:  "null_law"
;; [list]]
; Apply the pattern to the prism.
(define body (entity:pattern prism pat))
;; body
; OUTPUT Result
```

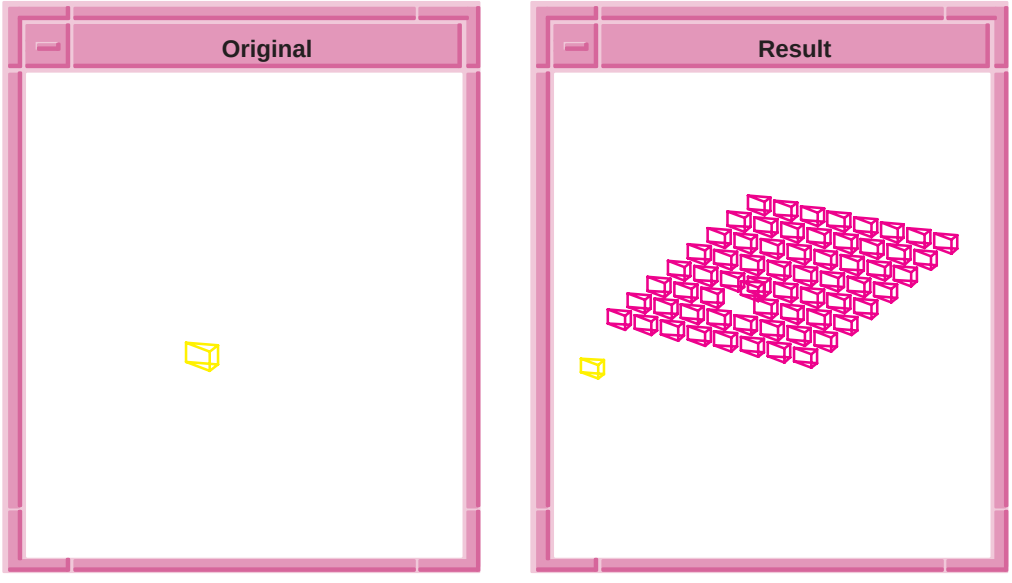

Figure 14-5. Moving Pattern Elements

pattern:periodic–keep

Scheme Extension: Patterns

Action: Creates a new pattern by applying a periodic keep filter to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_periodic_keep_pattern

Syntax: (**pattern:periodic–keep** pat keep which–dim [merge=#t])

Arg Types:

pat	pattern
keep	boolean
which–dim	integer
merge	boolean

Returns: pattern

Errors: None

Description: Applies a periodic keep filter to an existing pattern, merging with any existing filter or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

keep is the list of successive keep values, so that the size of the list is the periodicity of the filter.

which–dim specifies the dimension within which the filter is applied.

If merge is specified as #f, the existing pattern is replaced. The default value is #t.

Limitations: None

Example:

```
; pattern:periodic–keep
; 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 x–vec (gvector 4 0 0))
;; x–vec
(define num_x 6)
;; num_x
(define y–vec (gvector 0 2 0))
;; y–vec
(define num_y 12)
;; num_y
(define z–vec (gvector 0 0 3))
;; z–vec
(define num_z 4)
;; num_z
(define pat (pattern:linear x–vec num_x
    y–vec num_y z–vec num_z))
;; pat
; modify the pattern
(define keep1 #t)
;; keep1
(define keep2 #t)
;; keep2
(define keep3 #f)
;; keep3
(define dim 1)
;; dim
(set! pat (pattern:periodic–keep pat
    (list keep1 keep2 keep3) dim))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(4,0,0)+X2*VEC(0,2,0)
;; +X3*VEC(0,0,3),0,5,0,11,0,3)”
;; x–vec:   ”null_law”
;; y–vec:   ”null_law”
;; z–vec    ”null_law”
;; scale:   ”null_law”
;; keep:    ”null_law”
;; no list]]
; apply the pattern to the prism
```

```
(define body (entity:pattern prism pat))
;; body
```

pattern:periodic–scale

Scheme Extension: Patterns

Action: Creates a new pattern by applying a periodic scale to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_periodic_scale_pattern

Syntax:
```
(pattern:periodic–scale pat scales which–dim root
    [merge=#t])
```

Arg Types:

pat	pattern
scales	(real ...) | (gvector ...)
which–dim	integer
root	position
merge	boolean

Returns: pattern

Errors: None

Description: Applies a periodic scale to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

scales is the list of successive scale values, so that the size of the list is the periodicity of the scaling. All scale values in the list must be greater than zero. They may be given as real (for uniform scaling) or as gvectors (for non–uniform scaling).

which–dim specifies the dimension in which the scale is applied.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

If merge is specified as #f, the existing pattern is replaced. The default value is #t.

Limitations: None

Example:

```
; pattern:periodic-scale
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 8)
;; num_x
(define y-vec (gvector 0 2 0))
;; y-vec
(define num_y 10)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; modify the pattern
(define scale1 0.5)
;; scale1
(define scale2 1.5)
;; scale2
(define scale3 1.0)
;; scale3
(define scale4 2.0)
;; scale4
(define dim 0)
;; dim
(define root origin)
;; root
(set! pat (pattern:periodic-scale pat
```

```
    (list scale1 scale2 scale3 scale4) dim root))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)"
;; x-vec    "null_law"
;; y-vec:   "null_law"
;; z-vec:   "null_law"
;; scale:   "null_law"
;; keep:    "null_law"
;; no list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:planar?

Scheme Extension: Patterns

Action: Determines whether or not a pattern is parallel to a given plane.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:planar?** pat {face | {root-point normal}})

Arg Types:

pat	pattern
face	face
root–point	position
normal	gvector

Returns: boolean

Errors: None

Description: This extension returns #t if a pattern lies on the plane. Otherwise, it returns #f.

pat specifies a pattern.

face specifies a plane.

root–point specifies a point on the plane.

normal specifies the normal to the plane.

Limitations: None

Example:

```
; pattern:planar?
; define root position for test
(define root (position 0 0 3))
;; root
; make a linear pattern in the xy plane
(define x–vec (gvector 10 0 0))
;; x–vec
(define y–vec (gvector 10 10 0))
;; y–vec
(define pat (pattern:linear x–vec 3 y–vec 5))
;; pat
; Is it planar with respect to a normal in the
; x direction?
(pattern:planar? pat root
    (gvector:cross x–vec y–vec))
;; #t
; make a circular pattern in the xy plane
(define center (position 5 6 3))
;; center
(define normal (gvector 0 0 1))
;; normal
(set! pat (pattern:elliptical center normal 5))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(10,0,0)+X2*VEC
;; (10,10,0),0,2,0,4)”
;; x–vec:  ”null_law”
;; y–vec:  ”null_law”
;; z–vec:  ”null_law”
;; scale:  ”null_law”
;; keep:   ”null_law”
;; no list]
; Is it planar with respect to a normal in the
; z direction?
(pattern:planar? pat root normal)
;; #t
```

pattern:polar–grid

Scheme Extension: Patterns

Action: Creates a polar–grid pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_polar_grid_pattern

Syntax:

```
(pattern:polar-grid center normal num dist
    [start] [not-rotate=#f] [hex-symmetry=#f]
    [start-angle=0] [end-angle=360] [ratio=1])
```

Arg Types:

center	position
normal	gvector
num	integer
dist	real
start	gvector
not–rotate	boolean
hex–symmetry	boolean
start–angle	real
end–angle	real
ratio	real

Returns: pattern

Errors: None

Description: Creates a two–dimensional polar–grid pattern. The pattern coordinates are specified in the order (radial, angular).

center defines the root position (that may or may not lie upon the seed entity).

normal vector sets the orientation of the pattern.

num specifies the number of rings in the grid (including the center).

dist specifies the distance between the rings.

start is an optional argument that specifies the direction of the first spoke of the pattern.

If not–rotate is true (#t), the elements of the pattern are kept in a fixed relative orientation.

If hex–symmetry is true (#t), the hexagonal symmetry is maintained for patterns extending either 360 or 180 degrees.

start–angle and end–angle arguments fix the angular extent of the pattern.

ratio argument sets the ratio of minor/major radii at the pattern perimeter.

Limitations: None

Example:

```
; pattern:polar–grid
; 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_rings 5)
;; num_rings
(define spacing 4)
;; spacing
(define pat (pattern:polar–grid
    center normal num_rings spacing))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:print–transform

Scheme Extension: Patterns

Action: Prints the requested pattern transform to stdout.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: `(pattern:print-transform [index1 [index2]])`

Arg Types:

pat	pattern
index1	integer
index2	integer

Returns: boolean

Errors: None

Description: This extension prints a transform associated with the pattern specified by pat. By default, the root transformation is printed.

pat specifies a pattern.

If index1 is given, the relative transformation for the pattern element indexed by its value is output instead.

If both index1 and index2 are given, the output is the element which is indexed by index2 from the element indexed by index1.

Limitations: None

Example:

```
; pattern:print-transform
; make a pattern
(define axis_start (position 0 -25 0))
;; axis_start
(define axis_end (position 0 25 0))
;; axis_end
(define start-dir (gvector 1 0 0))
;; start-dir
(define radius 5)
;; radius
(define thread_distance 10)
;; thread_distance
(define edge (edge:helix axis_start axis_end
    start-dir radius thread_distance))
;; edge
(define num 50)
;; num
(define root (position 1 2 3))
;; root
(define on-endpoints #f)
;; on-endpoints
(define rail-dir (gvector 1 0 0))
;; rail-dir
(define tangent-dir (gvector 0 1 0))
;; tangent-dir
(define pat (pattern:edge edge num
    root on-endpoints rail-dir tangent-dir))
;; pat
(entity:delete edge)
;; ()
; print the root transform
(define result1 (pattern:print-transform pat))
;; result1
; print the first transform
(define result2 (pattern:print-transform pat 0))
;; result2
; check for success
(law:equal-test result1 #t)
;; #t
(law:equal-test result2 #t)
;; #t
```

pattern:radial

Scheme Extension: Patterns

Action: Creates a radial pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_radial_pattern

Syntax:
```
(pattern:radial center normal num–radial
    num–angular dist [start] [not–rotate=#f]
    [start–angle=0] [end–angle=360] [ratio=1])
```

Arg Types:

center	position
normal	gvector
num–radial	integer
num–angular	integer
dist	real
start	gvector
not–rotate	boolean
start–angle	real
end–angle	real
ratio	real

Returns: pattern

Errors: None

Description: Creates a two–dimensional radial pattern. The pattern coordinates are specified in the order (radial, angular).

center defines the root position (that may or may not lie upon the seed entity).

normal vector sets the orientation of the pattern.

num–radial specifies the number of elements in the radial direction.

num–angular specifies the number of elements in the angular direction.

dist specifies the distance between successive rings of the pattern.

start is an optional argument that specifies the direction of the first spoke of the pattern.

If not–rotate is true (#t), the elements of the pattern are kept in a fixed relative orientation.

start–angle and end–angle arguments fix the angular extent of the pattern.

ratio argument sets the ratio of minor/major radii at the pattern perimeter.

Limitations: None

Example:

```
; pattern:radial
; 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
```

pattern:random

Scheme Extension: Patterns

Action: Creates a random pattern within the indicated region.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_random_pattern

Syntax:
```
(pattern:random extents number [dimension=3]
    [ellipsoidal=#f] [x–vec=(1 0 0)] [y–vec=(0 1 0)])
```

Arg Types:

extents	gvector
number	integer
dimension	integer
ellipsoidal	boolean
x–vec	gvector
y–vec	gvector

Returns: pattern

Errors: None

Description: Creates a random pattern of number elements, centered at the location of the pattern seed entity and extending distances given by the components of extents in dimension dimensions.

extents specifies the extension distance.

number specifies the number of elements.

dimension specifies the extension dimension.

If ellipsoidal is set to true (#t), the number of pattern elements actually generated may differ somewhat from number.

x–vec and y–vec arguments specify the orientation of the pattern, and are the directions associated with the first two components of extents. (The third component is associated with the cross product of these arguments).

Limitations: None

Example:

```
; pattern:random
; 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 extents (gvector 50 25 10))
;; extents
(define num 100)
;; num
(define dim 3)
;; dim
(define pat (pattern:random extents num dim))
;; pat
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:random–keep

Scheme Extension: Patterns

Action: Creates a new pattern by applying a random keep filter to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_random_keep_pattern

Syntax: (**pattern:random–keep** pat fraction [merge=#t])

Arg Types:	pat	pattern
	fraction	real
	merge	boolean

Returns: pattern

Errors: None

Description: Applies a periodic keep filter to an existing pattern, merging with any existing filter or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

fraction determines the fraction of pattern elements that are kept.

If merge is specified as false (#f), the existing pattern is replaced. The default value is true (#t).

Limitations: None

Example:

```
; pattern:random-keep
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 12)
;; num_x
(define y-vec (gvector 0 4 0))
```

```
;; y–vec
(define num_y 12)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; modify the pattern
(define keep_fraction 0.5)
;; keep_fraction
(set! pat (pattern:random–keep pat keep_fraction))
;; #[pattern
;;  trans–vec:"DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,4,0),0,11,0,11)"
;; x–vec:   "null_law"
;; y–vec:   "null_law"
;; z–vec:   "null_law"
;; scale:   "null_law"
;; keep:    "null_law"
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:random–offset

Scheme Extension: Patterns

Action: Creates a new pattern by adding random offset to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_random_offset_pattern

Syntax: (**pattern:random–offset** pat amplitude)

Arg Types:

pat	pattern
amplitude	gvector

Returns: pattern

Errors: None

Description: Randomizes the given pattern by adding random offsets at each of the pattern sites.

pat specifies a pattern.

The components of the amplitude argument determine, in magnitude, the maximum offsets that are imposed in each dimension.

Limitations: None

Example:

```
; pattern:random-offset
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 12)
;; num_x
(define y-vec (gvector 0 4 0))
;; y-vec
(define num_y 12)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; modify the pattern
(define amplitude (gvector 1 0.5 4.0))
;; amplitude
(set! pat (pattern:random-offset pat amplitude))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(4,0,0)+X2*VEC(0,4,0),
;; 0,11,0,11)"
;; x-vec:   "null_law"
;; y-vec:   "null_law"
;; z-vec:   "null_law"
;; scale:   "null_law"
;; keep:    "null_law"
```

```
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:random–orient

Scheme Extension: Patterns

Action: Creates a new pattern by applying random rotations at each of the pattern sites.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_random_orient_pattern

Syntax: `(pattern:random–orient pat [min–axial max–axial axial–dir min–tilt max–tilt tilt–dir])`

Arg Types:

pat	pattern
min–axial	real
max–axial	real
axial–dir	gvector
min–tilt	real
max–tilt	real
tilt–dir	gvector

Returns: pattern

Errors: None

Description: Randomizes the given pattern by applying random rotations at each of the pattern sites. The default arguments yield a totally random rotation.

pat specifies a pattern.

min–axial and max–axial arguments specify the range of values in the axial direction specified by axial–dir.

axial–dir specifies the axial direction.

min–tilt and max–tilt arguments specify the range of values in the tilt direction specified by tilt–dir.

tilt–dir specifies the tilt direction. If the tilt–dir direction is not orthogonal to axial–dir, only its orthogonal component is used.

Limitations: None

Example:

```
; pattern:random-orient
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 8)
;; num_x
(define y-vec (gvector 0 2 0))
;; y-vec
(define num_y 10)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; modify the pattern
(define root origin)
;; root
(define min-axial 0)
;; min-axial
(define max-axial 360)
;; max-axial
(define axial-dir (gvector 1 0 0))
;; axial-dir
(define min-tilt 0)
;; min-tilt
(define max-tilt 0)
;; max_tiilt
(define tilt-dir (gvector 0 0 1))
```

```
;; tilt-dir
(set! pat (pattern:random-orient pat root
    min-axial max-axial axial-dir
    min-tilt max-tilt tilt-dir))
;; #[pattern
;; trans-vec:  DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)"
;; x-vec:   "null_law"
;; y-vec:   "null_law"
;; z-vec:   "null_law"
;; scale:   "null_law"
;; keep:    "null_law"
;; no list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:random–scale

Scheme Extension: Patterns

Action: Creates a new pattern by applying a random scale to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_random_scale_pattern

Syntax:
```
(pattern:random-scale pat min_scale max_scale
    root [merge=#t])
```

Arg Types:		
	pat	pattern
	min_scale	real | gvector
	max_scale	real | gvector
	root	position
	merge	boolean

Returns: pattern

Errors: None

Description: Applies a random scale to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

min_scale and max_scale arguments specify limits for the scale values. Both min_scale and max_scale must be greater than zero. They may be given as gvectors when non–uniform scaling is required.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

If merge is specified as false (#f), the existing pattern is replaced. The default vaule is true (#t).

Limitations: None

Example:

```
; pattern:random-scale
; 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 x-vec (gvector 4 0 0))
;; x-vec
(define num_x 8)
;; num_x
(define y-vec (gvector 0 2 0))
;; y-vec
(define num_y 10)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; modify the pattern
(define min_scale 0.5)
;; min_scale
(define max_scale 2.0)
;; max_scale
```

```
(define root origin)
;; root
(set! pat (pattern:random-scale
    pat min_scale max_scale root))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(4,0,0)+X2*VEC
;; (0,2,0),0,7,0,9)"
;; x-vec:  "null_law"
;; y-vec:  "null_law"
;; z-vec:  "null_law"
;; scale:  "null_law"
;; keep:   "null_law"
;; no list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:reflect

Scheme Extension: Patterns

Action: Creates a new pattern that is a reflection of the specified one through a given plane.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: **(pattern:reflect** pat root normal)

Arg Types:

pat	pattern
root	position
normal	gvector

Returns: pattern

Errors: None

Description: Creates a pattern that is a reflection of the pattern pat.

pat specifies a pattern.

root specifies a point on the plane of reflection.

normal specifies the normal to the reflection plane.

Limitations: None

Example:

```
; pattern:reflect
; 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 copy of the prism
(define prismr (entity:copy prism))
;; prismr
; make a pattern
(define t1 (transform:translation (gvector 0 0 0)))
;; t1
(define t2 (transform:translation (gvector 2 2 0)))
;; t2
(define t3 (transform:translation (gvector 4 0 0)))
;; t3
(define t4 (transform:translation (gvector 0 4 0)))
;; t4
(define t5 (transform:translation (gvector 4 4 0)))
;; t5
(define pat (pattern:from-list
    (list t1 t2 t3 t4 t5)))
;; pat
; reflect the pattern about the origin
(define root (position 0 0 0))
;; root
(define normal (gvector 1 1 0))
;; normal
(define patr (pattern:reflect pat root normal))
;; patr
; apply the original pattern to the original prism
```

```
(define apply (entity:pattern prism pat))
;; apply
; apply the reflected pattern to the copy
(define body (entity:pattern prismr patr))
;; body
```

pattern:remove

Scheme Extension: Patterns

Action: Removes all patterns from an entity.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:remove** entity)

Arg Types:	entity	entity

Returns: boolean

Errors: None

Description: Refer to Action.

entity specifies an entity.

Limitations: None

Example:

```
; pattern:remove
; create a pattern to be removed
(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 t1 (transform:translation (gvector 0 0 0)))
;; t1
(define t2 (transform:translation (gvector 2 2 0)))
;; t2
(define t3 (transform:translation (gvector 4 0 0)))
;; t3
(define t4 (transform:translation (gvector 0 4 0)))
;; t4
(define t5 (transform:translation (gvector 4 4 0)))
;; t5
(define pat (pattern:from–list
    (list t1 t2 t3 t4 t5)))
;; pat
; mirror the pattern about the origin
(define root (position 0 0 0))
;; root
(define normal (gvector 1 1 0))
;; normal
(set! pat (pattern:mirror pat root normal))
;; #[pattern [list]]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
; remove the pattern
(entity:remove body)
;; #t
```

pattern:remove–element

Scheme Extension: Patterns

Action: Removes a single element from an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:remove–element** pat coords)

Arg Types:		
	pat	pattern
	coords	pair

Returns: pattern

Errors: None

Description: Removes the element specified by coords from an existing pattern.

pat specifies a pattern.

coords identify the element in the pattern to be removed.

Limitations: None

Example:

```
; pattern:remove-element
; 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
```

```
; remove the center element
(set! pat (pattern:remove-element
    pat (list 0 0)))
;; #[pattern
;; trans-vec:  "DOMAIN(VEC(-1,0,0)*COS
;; (1.256637061435917*X2)*3*X+VEC
;; (0,-1,0)*SIN(1.256637061435917*X2)
;; *3*X+VEC(1,2,3),0,3,0,4)"
;; x-vec:  "VEC(-1,0,0)*COS
;; (1.256637061435917*X2)
;; +VEC(0,-1,0)*SIN
;; (1.25663706143591¿7*X2)"
;; y-vec:  "VEC(0,0,1)"
;; z-vec:  "null_law"
;; scale:  "null_law"
;; keep:   "PIECEWISE(X>0,X,X2<1)"
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:restore–element

Scheme Extension: Patterns

Action: Restores the element specified by coords to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:restore-element** pat coords)

Arg Types:

pat	pattern
coords	pair

Returns: pattern

Errors: None

Description: Restores the element specified by coords to an existing pattern.

pat specifies a pattern.

coords identify the element to be restored.

Limitations: None

Example:

```
; pattern:restore-element
; 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
; position the prism
(define origin (position 1 2 3))
;; origin
(define transform2 (entity:transform prism
    (transform:axes
    origin (gvector 1 0 0) (gvector 0 1 0))))
;; transform2
; make a pattern
(define x-vec (gvector 2 0 0))
;; x-vec
(define num_x 4)
;; num_x
(define y-vec (gvector 0 2 0))
;; y-vec
(define num_y 4)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; apply a keep law to the pattern
(define keep (law "X + Y < 4"))
;; keep
(set! pat (pattern:keep pat keep))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,2,0),0,3,0,3)"
```

```
;; x-vec:   "null_law"
;; y-vec:   "null_law"
;; z-vec:   "null_law"
;; scale:   "null_law"
;; keep:    "null_law"
;; no list]
; restore the corner element removed by the keep law
(set! pat (pattern:restore-element pat (list 3 3)))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,2,0),0,3,0,3)"
;; x-vec:   "null_law"
;; y-vec:   "null_law"
;; z-vec:   "null_law"
;; scale:   "null_law"
;; keep:    "X+Y<4"
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:scale

Scheme Extension: Patterns

Action: Creates a new pattern by applying a scale law to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:scale** pat scale root [merge=#t])

Arg Types:

pat	pattern
scale	law
root	position
merge	boolean

Returns: pattern

Errors: None

Description: Applies a scale law to an existing pattern, merging with any existing scaling or, optionally (with merge=#f), replacing it.

pat specifies a pattern.

scale law specifies the scaling to be applied. The scale law may evaluate to a scalar if unform scaling is desired or to a gvector if non–unform scaling is desired.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

If merge is specified as false (#f), the existing pattern is replaced. The default vaule is true (#t).

Limitations: None

Example:

```
; pattern:scale
; 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
; position the prism
(define origin (position 1 2 3))
;; origin
(define transform2 (entity:transform prism
    (transform:axes
    origin (gvector 1 0 0) (gvector 0 1 0))))
;; transform2
; make a pattern
(define x–vec (gvector 2 0 0))
;; x–vec
(define num_x 5)
;; num_x
(define y–vec (gvector 0 2 0))
```

```
;; y–vec
(define num_y 5)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; modify the pattern
(define scale (law ”(Y+1)/5”))
;; scale
(define root origin)
;; root
(set! pat (pattern:scale pat scale root))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,2,0),0,4,0,4)”
;; x–vec:  ”null_law”
;; y–vec:  ”null_law”
;; z–vec:  ”null_law”
;; scale:  ”null_law”
;; keep:  ”null_law”
;; no list]
; modify the pattern again
(set! scale (law ”(5 – X)/5”))
;; #[law ”(Y+1)/5”]
(set! pat (pattern:scale pat scale root))
;; #[pattern
;; trans–vec:  ”DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,2,0),0,4,0,4)+VEC(1,2,3)”
;; x–vec:  ”null_law”
;; y–vec:  ”null_law”
;; z–vec:  ”null_law”
;; scale:  ”(Y+1)/5”
;; keep:  ”null_law”
;; no list]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:scale–element

Scheme Extension: Patterns

Action: Scales a single element of an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:scale-element** pat coords scale root)

Arg Types:		
	pat	pattern
	coords	real
	scale	real | gvector
	root	position

Returns: pattern

Errors: None

Description: Scales the element specified by coords of an existing pattern about the position root by an amount scale.

pat specifies a pattern.

coords identify the element in the pattern.

scale argument specifies the scale value to be applied. The scale may be a real number, for uniform scaling, or a vector, for non–uniform scaling.

root specifies the neutral point about which the scaling takes place (i.e., the point on the seed entity that remains fixed while the entity's dimensions are altered).

Limitations: None

Example:

```
; pattern:scale-element
; 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 trans (entity:transform prism
    (transform:axes origin
    (gvector 1 0 0) (gvector 0 1 0))))
;; trans
```

```
; 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
; modify the pattern
(define scaling 2)
;; scaling
(define root origin)
;; root
(set! pat (pattern:scale–element pat
    (list 0 0) scaling root))
;; #[pattern
;; trans–vec:  "DOMAIN(VEC(–1,0,0)*COS
;; (1.256637061435917*X2)*3*X+VEC
;; (0,–1,0)*SIN(1.256637061435917*X2)
;; *3*X+VEC (1,2,3),0,3,0,4)"
;;  x–vec: "VEC(–1,0,0)*COS
;; (1.256637061435917*X2)+VEC
;; (0,–1,0)*SIN(1.256637061435917*X2)"
;; y–vec:  "VEC(0,0,1)"
;; z–vec:  "null_law"
;; scale:  "null_law"
;; keep:   "PIECEWISE(X>0,X,X2<1)"
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
(law:equal–test (length (entity:vertices body)) 96)
;; #t
```

pattern:set–root–transf

Scheme Extension: Patterns

Action: Creates a new pattern by applying a root transformation to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:set–root–transf** pat root–transf)

Arg Types:

pat	pattern
root–transf	transform

Returns: pattern

Errors: None

Description: Applies a root transformation to an existing pattern, replacing the existing root transformation, if any.

pat specifies a pattern.

root–transf specifies the root transformation to be applied.

Limitations: None

Example:

```
; pattern:set–root–transf
; make a body and a copy
(define body1 (solid:sphere (position 0 0 0) 1))
;; body1
(define body2 (entity:copy body1))
;; body2
; make a linear pattern along the x–axis
(define pat1 (pattern:linear (gvector 5 0 0) 5))
;; pat1
; apply the pattern to the original body
(define apply (entity:pattern body1 pat1))
;; apply
; make a new pattern from the original, but
; translated four units in the y direction
(define root–transf (transform:translation
    (gvector 0 4 0)))
;; root–transf
(define pat2 (pattern:set–root–transf
    pat1 root–transf))
;; pat2
; apply the new pattern to the second body
(define apply2 (entity:pattern body2 pat2))
;; apply2
```

pattern:size

Scheme Extension: Patterns

Action: Returns the number of elements in the given pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:size** pat)

Arg Types: pat pattern

Returns: integer

Errors: None

Description: This extension returns the number of elements in the pattern specified by pat.

Limitations: None

Example:

```
; pattern:size
; make a pattern
(define center (position 1 2 3))
;; 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
; find the pattern size
(define size (pattern:size pat))
;; size
```

pattern:spherical

Scheme Extension: Patterns

Action: Creates a spherical pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_spherical_pattern

Syntax:
```
(pattern:spherical center num–latitudes
    root [spacing])
```

Arg Types:

center	position
num–latitudes	integer
root	position
spacing	real

Returns: pattern

Errors: None

Description: Creates a two–dimensional spherical pattern about the center position, with the seed pattern entity at one pole of the associated sphere. The pattern coordinates are specified in the order longitude, latitude.

center specifies the center of the spherical pattern.

num–latitudes specifies the number of latitudinal rings in the pattern. If num–latitudes is set to zero, this number is instead determined by the optional spacing parameter.

root specifies the root position of the pattern.

spacing is an optional argument specifying the distance between the elements. The spacing must be given if num–latitudes is zero.

Limitations: None

Example:

```
; pattern:spherical
; make a pyramid
(define height 0.5)
;; height
(define maj_rad 1)
;; maj_rad
(define min_rad 0.1)
;; min_rad
(define num_sides 3)
;; num_sides
(define pyramid
    (solid:pyramid height maj_rad min_rad num_sides))
;; pyramid
;position the pyramid
(define origin (position 1 2 3))
;; origin
(define transform (entity:transform pyramid
    (transform:axes
    origin (gvector 1 0 0) (gvector 0 1 0))))
;; transform
(zoom–all)
;; #[view 1508128]
; output original

; make a pattern
(define center (position 1 2 –2))
;; center
(define num–latitudes 10)
;; num–latitudes
(define root origin)
;; root
(define pat
    (pattern:spherical center num–latitudes root))
;; pat
; apply the pattern to the pyramid
(define body (entity:pattern pyramid pat))
;; body
; output Result
```

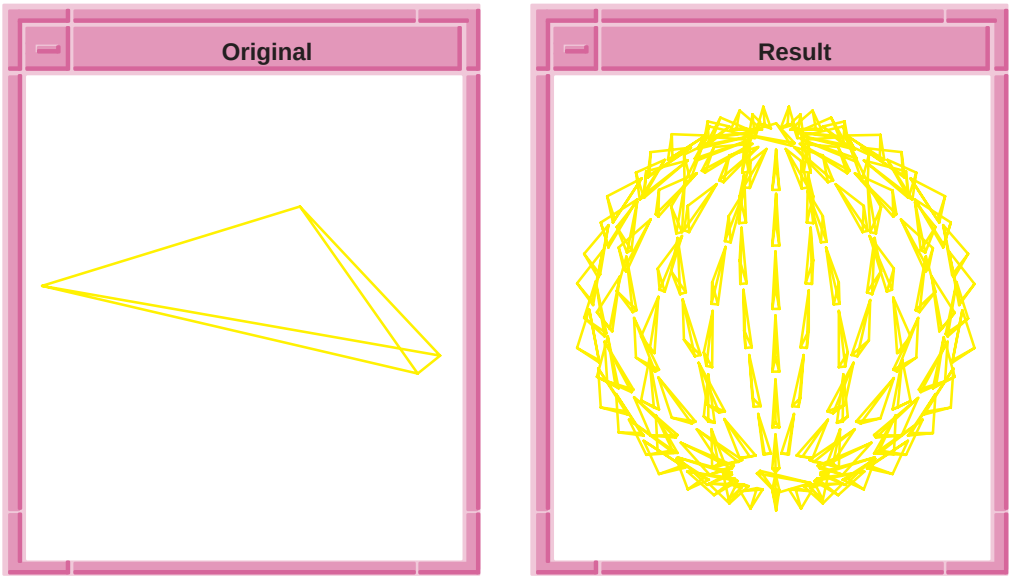

Figure 14-6. Spherical Patterns

pattern:spherical?

Scheme Extension: Patterns

Action: Determines whether or not a pattern lies on a given sphere.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: `(pattern:spherical? pat {face | center})`

Arg Types:

pat	pattern
face	face
center	position

Returns: boolean

Errors: None

Description: This extension returns #t if a pattern lies on the spherical surface associated with the face or, alternately, with center. Otherwise, it returns #f.

pat specifies a pattern.

face specifies a spherical face.

center specifies the center point of a spherical surface.

Limitations: None

Example:

```
; pattern:spherical?
; make a spherical pattern
(define center (position 5 6 3))
;; center
(define root (position 0 0 0))
;; root
(define pat (pattern:spherical center 5 root))
;; pat
; is it spherical about its center?
(pattern:spherical? pat center)
;; #t
; make a circular pattern in the xy plane
(define normal (gvector 0 0 1))
;; normal
(set! pat (pattern:elliptical center normal 5))
;; #[pattern
;; trans-vec:  "VEC(5,6,3)"
;; x-vec:  "DOMAIN(VEC(0,0.4472135954999579,
;; -0.8944271909999159)*COS
;; (0.89759790102¿56552*X)*COS
;; (0.7853981633974483*X2)+VEC
;; (0.801783725737273,
;; -0.5345224838248487,
;; -0.2672612419124243)*SIN
;; (0.8975979010256552*X)*COS
;; (0.7853981633974483*X2)+VEC
;; (0.5976143046671968,
;; 0.7171371656006361,
;; 0.3585685828003181)*SIN
;; (0.7853981633974483*X2),0,6,0,4)"
;; y-vec:  "VEC(0,0.4472135954999579,
;; -0.8944271909999159)*-1*SIN
;; (0.897597901025655¿2*X)+VEC
;; (0.801783725737273,
;; -0.5345224838248487,
;; -0.2672612419124243)*COS
;; (0.8975979010256552*X)"
;; z-vec:  "null_law"
```

```
;; scale:   ”null_law”
;; keep:    ”X2>0ANDX2<4ORNOT(X2>0ANDX2<4)ANDX=0”
;; no list]
; is it spherical about its center?
(pattern:spherical? pat center)
;; #t
; make a circular pattern in the yz plane
(set! pat
    (pattern:elliptical center (gvector 1 0 0) 5))
;; #[pattern
;; trans-vec:  ”VEC(5,6,3)”
;; x-vec:      ”DOMAIN(VEC(-1,0,0)*COS
;; (1.256637061435917*X)+VEC
;; (0,-1,0)*SIN(1.256637061435917*X),
;; 0,4)”
;; y-vec:      ”VEC(0,0,1)”
;; z-vec:      ”null_law”
;; scale:      ”null_law”
;; keep:       ”null_law”
;; no list]
;is it spherical about its center?
(pattern:spherical? pat center)
;; #t
; make a spherical pattern about a new center
(define new_center (position 5 5 3))
;; new_center
(set! pat (pattern:spherical new_center 5 root))
;; #[pattern
;; trans-vec:  ”VEC(5,6,3)”
;; x-vec:  ”DOMAIN(VEC(0,-1,0)*COS
;; (1.256637061435917*X)+VEC
;; (0,0,-1)*SIN(1.25663706¿1435917*X),
;; 0,4)”
;; y-vec:  ”VEC(1,0,0)”
;; z-vec:  ”null_law”
;; scale:  ”null_law”
;; keep:   ”null_law”
;; no list]
; is it spherical about the old center?
(pattern:spherical? pat center)
;; #f
```

pattern:surface

Scheme Extension: Patterns

Action: Creates a pattern parallel to a surface.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: api_surface_pattern

Syntax:
```
(pattern:surface entity num–u num–v root
    [on–boundary=#f] [dir–u dir–v])
```

Arg Types:

entity	entity
num–u	integer
num–v	integer
root	position
on–boundary	boolean
dir–u	gvector
dir–v	gvector

Returns: pattern

Errors: None

Description: Refer to Action.

entity specifies a face entity.

num–u and num–v arguments specify the number of elements along u and v directions respectively.

root specifies the position (which can be on or off the seed pattern entity, as desired) to be mapped to the pattern sites.

If on–boundary is set to true (#t), the pattern extends to the face boundary.

If dir–u and dir–v are specified, the orientation of the pattern members follow the surface normal. In this case, these vectors specify the directions, relative to the seed entity, that are mapped to the u– and v–directions of the face. By default, pattern members are oriented identically to one another.

Limitations: None

Example:
```
; pattern:surface
; 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 pos origin)
;; pos
(define radius 20)
;; radius
(define long_start 0)
;; long_start
(define long_end 90)
;; long_end
(define lat_start –360)
;; lat_start
(define lat_end 360)
;; lat_end
(define normal (gvector 0 0 1))
;; normal
(define face (face:sphere pos radius
    long_start long_end lat_start lat_end normal))
;; face
(define u_num 8)
;; u_num
(define v_num 6)
;; v_num
(define root origin)
;; root
(define pat (pattern:surface face
    u_num v_num root))
;; pat
(entity:delete face)
;; ()
```

```
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:transform

Scheme Extension: Patterns

Action: Creates a new pattern by applying a transformation to an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:transform** pat transf)

Arg Types:		
	pat	pattern
	transf	transform

Returns: pattern

Errors: None

Description: Applies the transformation transf to an existing pattern pat.

Limitations: None

Example:

```
; pattern:transform
; 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 x–vec (gvector 2 0 0))
;; x–vec
(define num_x 5)
;; num_x
(define y–vec (gvector 1 3 0))
;; y–vec
(define num_y 4)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; transform the pattern
(define t (transform:axes (position 4 5 6)
    (gvector 0 1 0) (gvector 0 0 1)))
;; t
(set! pat (pattern:transform pat t))
;; #[pattern
;; trans–vec:  "TRANS(DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (1,3,0),0,4,0,3),TRANS1)"
;; x–vec:      "DOMAIN(VEC(0,1,0),0,3)"
;; y–vec:      "DOMAIN(VEC(0,0,1),0,3)"
;; z–vec:      "null_law"
;; scale:      "null_law"
;; keep:       "null_law"
;; no list]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern:undo–move–element

Scheme Extension: Patterns

Action: Undoes a move applied to a single element of an existing pattern.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern:undo–move–element** pat coords)

Arg Types:		
	pat	pattern
	coords	pair

Returns: pattern

Errors: None

Description: Undoes all moves applied to the pattern element specified by coords, restoring it to its original position within the pattern.

pat specifies a pattern.

coords identifies the element in the pattern.

Limitations: None

Example:

```
; pattern:undo-move-element
; 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 x-vec (gvector 2 0 0))
;; x-vec
(define num_x 8)
;; num_x
(define y-vec (gvector 0 3 0))
;; y-vec
(define num_y 8)
;; num_y
(define pat (pattern:linear x-vec num_x y-vec num_y))
;; pat
; move one of the pattern elements
(define disp (gvector 0.4 1.2 0.5))
;; disp
(define move (transform:translation disp))
```

```
;; move
(set! pat (pattern:move-element pat (list 3 2) move))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,3,0),0,7,0,7)"
;; x-vec:      "null_law"
;; y-vec:      "null_law"
;; z-vec:      "null_law"
;; scale:      "null_law"
;; keep:       "null_law"
;; [list]]
; undo the move
(set! pat (pattern:undo-move-element pat (list 3 2)))
;; #[pattern
;; trans-vec:  "DOMAIN(X*VEC(2,0,0)+X2*VEC
;; (0,3,0),0,7,0,7)"
;; x-vec:      "null_law"
;; y-vec:      "null_law"
;; z-vec:      "null_law"
;; scale:      "null_law"
;; keep:       "null_law"
;; [list]]
; apply the pattern to the prism
(define body (entity:pattern prism pat))
;; body
```

pattern?

Scheme Extension: Patterns

Action: Determines whether or not a Scheme object is of the pattern Scheme data type.

Filename: kern/kern_scm/pattern_scm.cxx

APIs: None

Syntax: (**pattern?** object)

Arg Types:	object	scheme–object

Returns: boolean

Errors: None

Description: This extension returns #t if the given object is a pattern Scheme data type. Otherwise, it returns #f.

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

Limitations: None

Example:

```
; pattern?
; Create prism geometry to illustrate command.
(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
(zoom–all)
;; [view 13369704]
; Verify prism is not a pattern.
(pattern? prism))
;; #f
; Create a pattern.
(define x–vec (gvector 2 0 0))
;; x–vec
(define num_x 5)
;; num_x
(define y–vec (gvector 0 2 0))
;; y–vec
(define num_y 5)
;; num_y
(define pat (pattern:linear x–vec num_x y–vec num_y))
;; pat
; Verify pat is a pattern.
(pattern? pat)
;; #t
```

point:position

Scheme Extension: Construction Geometry

Action: Computes the position of a point.

Filename: kern/kern_scm/pnt_scm.cxx

APIs: None

Syntax: `(point:position point)`

Arg Types: point — point

Returns: position

Errors: None

Description: This extension accepts an entity of the type point, and it returns the position of the point.

Limitations: None

Example:

```
; point:position
; Create a point.
(define point1 (point (position 6 6 7)))
;; point1
; Get the location of the point.
(point:position point1)
;; #[position 6 6 7]
```

point?

Scheme Extension: Model Geometry

Action: Determines if a Scheme object is a point.

Filename: kern/kern_scm/pnt_scm.cxx

APIs: None

Syntax: `(point? object)`

Arg Types: object — scheme–object

Returns: boolean

Errors: None

Description: Refer to Action.

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

Limitations: None

Example:

```
; point?
; Create a point.
(define point1 (point (position 6 6 7)))
;; point1
; Determine if the point is actually a point.
(point? point1)
;; #t
```

position

Scheme Extension: Mathematics

Action: Creates a new position with coordinates *x*, *y*, and *z*.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position x y z [space="models"])`

Arg Types:

x	real
y	real
z	real
space	string

Returns: position

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 following values for the space argument are:

- "wcs" is the default if an active WCS exists. Otherwise, the default setting 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.

A position is not saved with the part, but is used to help define geometry. Positions are not displayed in Scheme. A point is an entity. A point is different from a vertex in that it has no edge associations. Use env:set–point–size and env:set–point–style to change its appearance.

Limitations: None

Example:

```
; position
; Create a position.
(define pos1 (position 3 3 3))
;; pos1
; Create a position in the
; "cylindrical" coordinate system.
(define pos-cyl (position 20 20 0 "cylindrical"))
;; pos-cyl
; Define a solid block using the created position.
(define block1 (solid:block pos1 pos-cyl))
;; block1
```

position:closest

Scheme Extension: Mathematics, Analyzing Models

Action: Gets the position from a list of positions that is closest to a given position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:closest** position position-list)

Arg Types:		
	position	position
	position–list	position | (position ...)

Returns: position

Errors: None

Description: Refer to Action.

position specifies the location that all other positions are compared against.

position–list specifies the locations to compare against the given position.

Limitations: None

Example:

```
; position:closest
; Determine the closest position in a list
; of positions to a given position.
(define pos1 (position:closest (position 30 30 30)
    (list (position 10 10 0) (position 25 22 0))))
;; pos1
```

position:copy

Scheme Extension: Mathematics

Action: Creates a new position by copying an existing position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:copy** position)

Arg Types:	position	position

Returns: position

Errors: None

Description: Refer to Action.

position specifies the position object that has to be copied.

Limitations: None

Example:

```
; position:copy
; Create a new position by copying
; an existing position.
(define pos1 (position 20 20 0 ”cylindrical”))
;; pos1
(define pos2 (position:copy pos1))
;; pos2
```

position:distance

Scheme Extension: Mathematics, Analyzing Models

Action: Gets the distance between two positions.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:distance position1 {position2 | ray})`

Arg Types:		
	position1	position
	position2	position
	ray	ray

Returns: real

Errors: None

Description: Refer to Action.

position1 defines the start location.

position2 defines the end location.

ray defines the ray to calculate the distance.

Limitations: None

Example:

```
; position:distance
; Determine the distance between two positions.
(define pos1 (position 20 20 0 ”cylindrical”))
;; pos1
(position:distance pos1 (position 0 20 0))
;; 22.9430574540418
```

position:interpolate

Scheme Extension: Mathematics

Action: Interpolates a new position based on the weight between two given positions.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:interpolate position1 position2 weight)`

Arg Types:		
	position1	position
	position2	position
	weight	real

Returns: position

Errors: None

Description: This extension returns the position as determined by weight.

position1 specifies the first position.

position2 specifies the second position.

weight is a real number, ranging from 0 to 1.

The position is given by:

```
((1–weight) * position1) + (weight * position2)
```

Given weight:	Resulting position:
0	position1
1	position2
0.5	The midpoint between position1 and position2.

Limitations: None

Example:

```
; position:interpolate
; Interpolate new positions between
; two given positions.
(position:interpolate (position 3 5 4)
    (position 3 3 3) 0)
;; #[position 3 5 4]
(position:interpolate (position 3 5 4)
    (position 3 3 3) 1)
;; #[position 3 3 3]
(position:interpolate (position 3 5 4)
    (position 3 3 3) 0.5)
;; #[position 3 4 3.5]
```

position:offset

Scheme Extension: Mathematics

Action: Creates a new position offset from a given position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:offset** position gvector)

Arg Types:		
	position	position
	gvector	gvector

Returns: position

Errors: None

Description: Refer to Action.

position specifies the location of the old position.

gvector specifies the gvector to apply to the position.

Limitations: None

Example:

```
; position:offset
; Create a new position, offset by a gvector
; from a given position.
(position:offset (position 0 0 0)
    (gvector 10 10 10))
;; #[position 10 10 10]
; Define the start position.
(define pos–start (position 0 –4 0))
;; pos–start
; Define the start direction.
(define dir–start (gvector 8 –8 0))
;; dir–start
; Create a new position, offset by a gvector
; from a given position.
(position:offset pos–start dir–start)
;; #[position 8 –12 0]
```

position:project–to–line

Scheme Extension: Mathematics, Projecting

Action: Gets the projection of a position on to a line.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:project–to–line** position line–pos line–dir)

Arg Types:		
	position	position
	line–pos	position
	line–dir	gvector

Returns: position

Errors: None

Description: line–pos and line–dir together define the line. This extension returns the position on the line.

position specifies the position to project.

line–pos specifies a position on the line.

line–dir specifies a gvector defining the direction of the line.

Limitations: None

Example:
```
; position:project-to-line
; Project a position onto a line.
(position:project-to-line (position 10 10 10)
    (position 0 0 0) (gvector 0 1 0))
;; #[position 0 10 0]
```

position:project–to–plane

Scheme Extension: Mathematics, Projecting

Action: Gets the projection of a position onto a plane.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax:
```
(position:project-to-plane position
    plane-pos plane-normal)
```

Arg Types:		
	position	position
	plane–pos	position
	plane–normal	gvector

Returns: position

Errors: None

Description: plane–pos and plane–normal together specify the plane. This extension returns the position on the plane.

position specifies the position to project.

plane–pos specifies a position on the plane.

plane–normal specifies a gvector defining the plane normal.

Limitations: None

Example:

```
; position:project-to-plane
; Project a position onto a plane.
(position:project-to-plane (position 0 0 0)
    (position 25 25 25) (gvector 1 0 0))
;; #[position 25 0 0]
```

position:set!

Scheme Extension: Mathematics

Action: Sets the *x*, *y*, and *z*–components of a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:set!** pos1 {{x y z} | {pos2}})

Arg Types:

pos1	position
x	real
y	real
z	real
pos2	position

Returns: position

Errors: None

Description: Refer to Action.

pos1 specifies the position to be set.

If x, y and z values are specified, they are copied into pos1.

If pos2 is specified, its position value is copied into pos1.

Limitations: None

Example:

```
; position:set!
; Set new x-, y-, and z-coordinates for
; an existing position.
(define pos1 (position 0 0 0))
;; pos1
(position:set! pos1 3 5 4)
;; #[position 3 5 4]
```

position:set–x!

Scheme Extension: Mathematics

Action: Sets the *x*–component of a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:set–x! position x)`

Arg Types:

position	position
x	real

Returns: real

Errors: None

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

position identifies the original *y* and *z* values.

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

Limitations: None

Example:

```
; position:set–x!
; Set a new x–coordinate for an existing position.
(define pos1 (position 0 0 0))
;; pos1
(position:set–x! pos1 5)
;; 5
```

position:set–y!

Scheme Extension: Mathematics

Action: Sets the *y*–component of a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:set–y! position y)`

Arg Types:

position	position
y	real

Returns: real

Errors: None

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

position identifies the original *x* and *z* values.

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

Limitations: None

Example:
```
; position:set–y!
; Set a new y–coordinate for an existing position.
(define pos1 (position 0 0 0))
;; pos1
(position:set–y! pos1 8)
;; 8
```

position:set–z!

Scheme Extension: Mathematics

Action: Sets the *z*–component of a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:set–z!** position z)

Arg Types:		
	position	position
	z	real

Returns: real

Errors: None

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

position specifies the original *x* and *y* values.

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

Limitations: None

Example:
```
; position:set–z!
; Set a new z–coordinate for an existing position.
(define pos1 (position 0 0 0))
;; pos1
(position:set–z! pos1 –4)
;; –4
```

position:transform

Scheme Extension: Mathematics, Transforms

Action: Applies a transform to a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:transform position transform)`

Arg Types:	position	position
	transform	transform

Returns: position

Errors: None

Description: Refer to Action.

position specifies the location to apply the transform.

transform is any valid transform.

Limitations: None

Example:

```
; position:transform
; Apply a transform to a position.
(define pos1 (position 5 10 15))
;; pos1
(define pos2 (position:transform pos1
    (transform:translation (gvector 0 1 0))))
;; pos2
```

position:x

Scheme Extension: Mathematics

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

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:x position)`

Arg Types:	position	position

Returns: real

Errors: None

Description: Refer to Action.

position specifies a position.

Limitations: None

Example:
```
; position:x
; Get a position's x-coordinate.
(define pos1 (position 5 10 15))
;; pos1
(position:x pos1)
;; 5
```

position:y

Scheme Extension: Mathematics

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

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: (**position:y** position)

Arg Types: position — position

Returns: real

Errors: None

Description: Refer to Action.

position specifies a position.

Limitations: None

Example:
```
; position:y
; Get a position's y-coordinate.
(define pos1 (position 5 10 15))
;; pos1
(position:y pos1)
;; 10
```

position:z

Scheme Extension: Mathematics

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

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position:z position)`

Arg Types:	position	position

Returns: real

Errors: None

Description: Refer to Action.

position specifies a position.

Limitations: None

Example:

```
; position:z
; Get a position's z–coordinate.
(define pos1 (position 5 10 15))
;; pos1
(position:z pos1)
;; 15
```

position?

Scheme Extension: Mathematics

Action: Determines if a Scheme object is a position.

Filename: kern/kern_scm/pos_scm.cxx

APIs: None

Syntax: `(position? 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 position.

Limitations: None

Example:

```
; position?
; Determine if a position is a position.
(define pos1 (position 5 10 15))
;; pos1
(position? pos1)
;; #t
(position? (position 6 5 8))
;; #t
; Determine if a gvector is a position.
(position? (gvector 0 0 0))
;; #f
; Determine if a value is a position.
(position? –4)
;; #f
; Define a new position.
(define pts (position 10 10 10))
;; pts
; Determine if the new position is a position.
(position? pts)
;; #t
```