An Excerpt of the CindyScript Documentation

Defining Functions

Defining a function in CindyScript is very easy. One has simply to define the name of a function, provide a parameter list, and write down the body of the function. No explicit typing of arguments or function values is required. The return value of a function is the value of the last evaluated statement in the function. For example,

```
f(n):=sum(1..n,i,i^2)
```

calculates the sum of the first n squares. For instance, after this definition, `f(4)` evaluates to 30.

Assigning Hash Values

To a point, an additional value can be assigned, which can also be read out. For example,

```
A:“flavour“ = [“chocolate“];
B:“birthday“ = [11,2,1983];
```

The points `A` and `B` carry the values flavour and birthday, respectively. They can be changed in the same way as above and can be read out by

```
A:“flavour“;
B:“birthday“;
```

Drawing Functions

`draw(<expr>)`: The `draw(<expr>)` function is a multifunctional operator. Depending on the meaning of `<expr>`, the corresponding objects will be drawn if possible.

Typing of <expr>:	Drawing Action:
`[<real1>,<real2>]`	Will draw a point with *x*-coordinate `<real1>` and *y*-coordinate `<real2>`
`[<point1>,<point2>]`	Will draw a segment from `<point1>` to `<point2>`

Example: The code below draws the points `A` and `B`, as well as the segment `AB`.

```
A=[0,0]; B=[1,0];
draw(A);
```

```
draw(B);
draw([A,B]);
```

`fillcircle(<point>,<real> )`: This function draws a filled circle with center `<point>` and radius `<real>`.

Example:

```
A=[0,0]; r=1;
filledcircle(A,1);
```

`drawcircle(<point>,<real>)`: This function draws a circle with center `<point>` and radius `<real>`.

`drawpoly(<list>)`: This operator takes a list of points as input and creates a polygon from them.

`connect(<list>)`: This operator takes a list of points as input and connects them by line segments.

Example: The following code draws the border of a square.

```
A=[0,0]; B=[1,0]; C=[1,1]; D=[0,1];
connect([A,B,C,D]);
```

Modifiers: The drawing operator can handle several modifiers. They are summarized in the following table:

Modifier:	type of value:	effect
`pointsize`	`<real>`	sets the point size
`linesize`	`<real>`	sets the line size
`size`	`<real>`	sets the line size and the point size
`pointcolor`	`[<real1>, <real2>, <real3>]`	sets the point color to an RGB value

Modifier:	type of value:	effect
`linecolor`	`[<real1>, <real2>, <real3>]`	sets the line color to an RGB value
`color`	`[<real1>, <real2>, <real3>]`	sets the point color and the line color to an RGB value
`alpha`	`<real>`	sets the opacity to the value `<alpha>`
`noborder`	`<bool>`	`noborder→true` turns off the border of points
`border`	`<bool>`	`border→true` the opposite of the last modifier

Modifiers have only a local effect. This means that the default appearance settings are not affected when a modifier is used.

Output

`print(<expr>)`**:** This operator prints the result of evaluating `<expr>` to the console.

`println(<expr>)`: This operator prints the result of evaluating `<expr>` to the console and adds a newline character to the end of the text.

`err(<expr>)`: Prints the result of evaluating `<expr>` to the console. If `<expr>` is a variable, the variable name is printed as well. Very useful for debugging.

Arithmetic and Trigonometric Functions

`^`: A number (integer, real, complex) can be taken to the power of another number (integer, real, complex).

`sin(_)`: Produces the sine of a number.

`cos(_)`: Produces the cosine of a number.

`exp(_)`: Produces the exponential function of a number.

`sqrt(_)`: Produces the square root of a number.

`arctan2(_,_)`: arctan2(x,y) is the angle of the vector (x,y).

`°`: This operator multiplies any number by the constant pi/180 . This makes possible angle conversion from degrees to radians.

Example: `180°` evaluates to `3.1416.....`

`random()`: The operator generates a random real number between 0 and 1.

List Operations

`1..n`: Produces the list of integers 1, 2, 3, ..., n.

`<list1> ++ <list2>`: This operator creates a list by concatenation of two other lists.

Example: `[„a“, „b“]++[„c“, „d“]` evaluates to `[„a“, „b“, „c“, „d“]`

`<list1> -- <list2>`: This operator creates a list by removing all elements that occur in `<list2>` from `<list1>`.

`forall(<list>,<expr>)`: This operator is useful for applying an operation to all elements of a list. It takes a `<list>` as first argument. It produces a loop in which `<expr>` is evaluated for each entry of the list. For each run, the run variable # takes the value of the corresponding list entry.

Example: `a=[„this“, „is“, „a“, „list“]; forall(a,println(#));` produces the output

```
this
is
a
list
```

`apply(<list>,<expr>)`: This operator generates a new list by applying the operation <expr> to all elements of a list and collecting the results. As usual, # is the run variable, which successively takes the value of each element in the list.

Example: `apply([1,2,3], #^2)` evaluates to `[1,4,9]`

`select(<list>,<boolexpr>)`: This operator selects all elements of a list for which a certain condition is satisfied. The condition is supposed to be encoded by `<boolexpr>`. This expression is assumed to return a `<bool>` value. As usual, # is the run variable, which successively takes the value of all elements in the list.

Example: `select([1,2,3], isodd(#))` evaluates to `[1,3]`

Note: There are also the commands `forall(<list>,<var>,<expr>)`, `apply(<list>,<var>,<expr>)` and `select(<list>,<var>,<expr>)`. They are similar to original commands, but the run variable is now named `<var>`.

`pairs(<list>)`: This operator produces a list that contains all two-element sublists of a list. These are all pairs of elements of `<list>`. This operator is particularly useful for creating all segments determined a set of points.

`sort(<list>)`: Within CindyScript, all elements possess a natural complete order that makes it possible to compare any two elements. Two elements are equal, or one of them is greater than the other. Within the real numbers, the order is the usual numeric ordering. Within strings, the order is the lexicographic order. Complex numbers are ordered by their real parts first. If two complex numbers have the same real part, then they are compared with respect to their imaginary parts. Two lists are compared by the first entry in which they differ. Furthermore, by convention we have the ordering

booleans < numbers < strings < lists

`sort(<list>,<expr>)`: This operator takes each element of the list and evaluates a function expressed by `<expr>` applied to it. All elements of the list are sorted with respect to the result of these evaluations.

Example: `sort([-1,2,-4], abs(#))` evaluates to `[-1,2,-4]`

Control Structures

`if(<bool>,<expr1>,<expr2>)`: The expression `<expr1>` will be evaluated if the Boolean condition `<bool>` evaluates to true. If `<bool>` evaluates to false, then `<expr2>` is evaluated. In each case the value of the evaluated expression is returned. Thus this ternary version of the if-operator encodes an if/then/else functionality. There are two typical uses of this version of the if-operator: First, the if-operator is used to force the conditional evaluation of program parts (which usually will cause side effects).

Example: This code fragment defines the function `f(x)` to be the absolute value function (for real values of x).

```
f(x):=if(x>0,x,-x)
```

`repeat(<number>,<expr>)`: This operator provides the simplest kind of loop in CindyScript: `<expr>` will be evaluated `<number>` times. The result of the last evaluation will be returned. During the evaluation of `<expr>` the special variable # will contain the counting variable of the loop.

Example: `repeat(6, println(# + „ “))` evaluates to `1 2 3 4 5 6`

List of Geometric Objects

`allpoints()`: Returns a list of all points.

`allmasses()`: Returns a list of all masses.

Function Plotting

`plot(<expr>)`: The plot operator can be used to plot a function. The function must be given as an expression `<expr>`. This expression must contain the running variable # and calculate either a real value for a real input of # or a two-dimensional vector. In the first case, the plot operator will simply draw the function. In the latter case, it will draw a parametric plot of a function. The coordinate system is tied to the coordinate system of the

geometric views. The plot operator also provides many different modifiers. Below, there are some listed.

Modifier:	Type of Value:	Effect:
`color`	`[<real1>,<real2>,<real3>]`	set color to specific value
`size`	`<real>`	set line size to specific value
`alpha`	`<real>`	set opacity
`start`	`<real>`	set start value for function drawing
`stop`	`<real>`	set end value for function drawing
`steps`	`<real>`	number of set plot points (for parametric functions only)
`pxlres`	`<real>`	pixel resolution of curve plotting (for real functions only)

Example: `plot(sin(#))` plots the sine, while `plot([cos(#),sin(#)])` plots the unit circle.

`colorplot(<expr>,<vec>,<vec>)`: The colorplot operator makes it possible to give a visualization of a planar function. To each point of a rectangle a color value can be assigned by a function. In the function <expr> the running variable is again #. However, it is important to notice that this variable describes now a point in the plane. The return value of <expr> should be either a real number (in which case a gray value is assigned) or a vector of three real numbers (in which case an RGB color value is assigned). In any case, the values of the real numbers should lie between 0 and 1. The second and third argument determine the lower left and the upper right corners of the drawing area.

`drawfield(<expr>)`: The drawfield operator can be used to draw a vector field. The function must be given as an expression `<expr>`. This expression must contain the running variable #, which should this time be a two-dimensional vector. The result should also be a two-dimensional vector. Applying the operator drawfield to this expression will result in plotting the corresponding vector field. The field will be animated. This means that it will change slightly with every picture. Therefore, it is often useful to put the drawfield operator into the "timer tick" evaluation slot. This creates an animation control in the geometric view. Running the animation will automatically animate the vector field. In the drawfield operator the run variable for the function `<expr>` must be the # variable.

Example: We consider a vector field defined by the function `f(x,y)=(y,sin(x))`. The corresponding code with the function definition and the call of the drawfield operator is as follows:

```
f(v):=[v.y,sin(v.x)];
drawfield(f(#));
```

Physics and CindyScript

Masses in CindyScript: A mass point provides several fields that can be read and generally set by CindyScript. The following list shows the accessible fields for masses:

- **vx**: *x*-component of velocity (real number, set and read)
- **vy**: *y*-component of velocity (real number, set and read)
- **v**: velocity as a vector (vector of two real numbers, set and read)
- **fx**: *x*-component of force (real number, read only)
- **fy**: *y*-component of force (real number, read only)
- **f**: force as a vector (vector of two real numbers, read only)
- **kinetic**: the kinetic energy of the point (real number, read only)
- **ke**: the kinetic energy of the point (real number, read only)
- **mass**: the mass of the point (real number, set and read)
- **friction**: the friction of the point (real number, set and read)
- **charge**: the charge of the point (integer number, set and read)
- **radius**: the radius of the point (real number, set and read)
- **simulate**: turn on/off simulation for this point (Boolean, set and read)

Mouse Position

mouse(): Returns a vector that represents the current position of the mouse if the mouse is pressed. The vector is given in homogeneous coordinates (this allows also for acces of inifite objects). If one needs the two-dimensional euclidean coordinates of the mouse position one can access them via `mouse().xy`.

Execution Times

Every script in Cinderella is associated with an occasion on which it will be executed. Since scripts in Cinderella are assumed to run at the runtime of the program, it may well happen that the same script is executed many times during a move. The precise interpretation of the different occasions is described below:

- **Draw:** A script present in this slot will be executed right before a new screen picture is generated. Scripts entered here will be executed very often. This is the typical place to enter a script that should be automatically updated during a drag of a geometric elements.

- **Move:** Scripts in this slot will be called even more often than those in the "Draw" slot. They will always be invoked if internally the position of the free elements has changed. In general, this is more often than at screen refresh. Sometimes putting a script in the "Draw" slot produces strange and unexpected results. It might then be good to place it in the "Move" slot.

- **Initialization:** A script in this slot is executed whenever in the program the parse button of the script editor is pressed. This slot it is very useful for resetting variables or setting points to an initial position.

- **Timer Tick:** If a script is contained in this slot it will cause an animation controller to be shown in geometric view (if not already present). When the play button of the animation controller is pressed, a script in this slot will be regularly executed every few milliseconds.

- **Simulation Start:** This script is executed when the animation controller changes from "Stop" to "Play." It is a very good place to enter initial setups for animations.

- **Simulation Stop:** This script is executed when the animation controller changes from "Play" to "Stop." It is a very good place to evaluate the result of an animation.

- **Mouse Down:** This slot and the following three slots are very useful for programming under interfaces in CindyScript. A script in this slot is executed whenever the mouse button is pressed. With the function mouse() the current mouse coordinates can be read within a script.

- **Mouse Up:** Scripts in this slot will be executed when the mouse is released.

- **Mouse Click:** Scripts in this slot will be executed when the mouse is moved.

- **Mouse Drag:** Scripts in this slot will be executed when the mouse is dragged.

- **Key Typed:** Scripts in this slot will be executed when the key on the keyboard is pressed. This slot is very useful for handling user input via the keyboard. A string that corresponds to the currently pressed key can be accessed via the key() function.