`R`	`displacement`	`llimit`	`psurf`	`stiff`
`T`	`dullness`	`local`	`quat`	`stiffness`
`ambient`	`edge`	`localforce`	`radius`	`texture`
`archive`	`face`	`localmoment`	`refraction`	`texture`
`asurf`	`figure`	`location`	`rest`	`to`
`attribute`	`globalforce`	`mass`	`rgb`	`tolerance`
`attributes`	`globalmoment`	`maxtrans`	`root`	`trans`
`blur`	`glossiness`	`mintrans`	`scale`	`transexp`
`concentration`	`hsv`	`moment`	`segment`	`transparency`
`connect`	`include`	`node`	`shademode`	`type`
`constraint`	`inertia`	`nstrength`	`site`	`ulimit`
`damp`	`intensity`	`patch`	`specular`	`vrpd`
`density`	`joint`	`path`	`sphere`	`xyz`
`diffuse`	`light`	`pstrength`	`stexture`	

Table A.1: Peabody Keywords

The peabody construct declarations create objects implicitly, not explicitly. Figures, segments, sites, and joints are created when they are defined for the first time. In this sense, the "definitions" of the constructs are not really *definitions* by *declarations.* It is entirely legal to have duplicate declarations for the same object. You should think of the process of reading a peabody file as a sequential stream of assignment statements which set the values of object parameters. The objects are created when they are first referenced. Duplicate assignment statements reset the parameter values.

A.3.1 Figure Declarations

A figure declaration begins with the keyword `figure`, followed by an identifier. A figure declaration may not occur inside any other block. Inside the figure block there may be:

- segment declarations
- site declarations
- joint declarations
- light declarations
- attribute declarations
- posture declarations

Site declarations must have names qualified with the segment name unless they are contained in a segment block. In addition, figures have the following assignment fields:

archive The archive specifies an archive file in which to look for psurf files for the segments in the figure. This field is optional, but if it is used it should occur before any segments which reference psurfs in the archive. The value is a character string giving the name of the archive file.

root The root specifies the root site for a figure. This is the site with respect to which the figure's `location` field describes its global placement. The value is the name of the site.

location The location gives the transform which defines the global placement of the figure's root. The placement of all other sites and segments in the figure is determined from this. The value is a homogeneous transformation.

A.3.2 Segment Declarations

A segment declaration begins with the keyword **`segment`**, followed by an identifier. A segment declaration may occur inside a figure block. If it lies outside a figure block, its name must be qualified with a figure name. Site declarations and attribute definitions may occur inside the segment block.

In addition, segments have the following assignment fields:

psurf The psurf gives the file which describes the geometry of the segment. The value is a character string which is the name of the psurf file.

attributes The set of attributes describes which surface attributes are to be associated with the psurf. The value is an array of attribute names, separated by commas. The number of attributes in this array must match the number of attribute indices used in the psurf file. If the psurf has a single attribute, then the value may be just the attribute name, and you may use the keyword **`attribute`**, without the 's'.

Peabody has a built-in set of rules for where it looks for psurf files. See Section 9.3 for a description of these rules. See Section A.4 for a description of the syntax of the psurf files.

A.3.3 Path Declarations

A *path* is a set of n sites on a segment, named $\{ pnt_0, pnt_1, \ldots pnt_n \}$, with one additional site called *point*. The peabody description of a path looks like:

```
segment paths {
    site base->location = trans(0cm,0cm,0cm);
    site point->location = trans(0cm,0cm,0cm);
    site pnt0->location = trans(0cm,0cm,0cm);
    site pnt1->location = xyz(0deg,0deg,-38deg) * trans(3cm,62cm,-11cm);
    site pnt2->location = xyz(-180deg,-75deg,141deg) * trans(45cm,62cm,-86cm);
    site pnt3->location = xyz(-180deg,-10deg,141deg) * trans(112cm,62cm,-1cm);
    site pnt4->location = trans(136cm,62cm,75cm);
    path = ("spline", 31, (site)point,
        (site)pnt0, 0,
        (site)pnt1, 0.25,
        (site)pnt2, 0.50,
        (site)pnt3, 0.75,
        (site)pnt4, 1);
}
```

The path field of the segment defines the interpolation method ("spline"), the number of generated samples (default 31), the point site, and the set of sites with their corresponding time values. The point site is used when animating a path, and the point site moves to a position on the path corresponding to a time fraction between 0 and 1 (i.e. the spline is evaluated at some time between 0 and 1, and the point site is moved to that location.

A.3.4 Site Declarations

A site declaration begins with the keyword **`site`**. A site declaration may occur inside a segment block. If it lies outside, its name must be fully qualified with the segment name. If it does not lie within a figure block, the name must also be qualified with the figure name.

A site has a single following assignment field, giving its location.

location The location is a transform describing the position and orientation of the site with respect to the coordinate frame of its segment. The value is a homogeneous transformation.

A.3.5 Joint Declarations

The declaration of a joint begins with the keyword `joint`. A joint connects two sites on two segments within the same figure, and represents a transformation between segments.

The principle statement in a joint declaration is the `connect` statement, which is not in the form of an assignment. It specifies the two sites which the joint connects, and it takes the form of "`connect site1 to site2`"

The joint has the following assignment fields:

- **type** A functional expression describing the degrees of freedom of the joint. The value of the expression is described below.
- **displacement** A vector of numbers specifying the current angles of the joint, having as many elements as the joint has degrees of freedom.
- **ulimit** A vector of numbers describing the upper limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.
- **llimit** A vector of numbers describing the lower limit of the angle or distance of each degree of freedom, having as many elements as the joint has degrees of freedom.

By default, the transformation at a joint is an arbitrary homogeneous transformation, which specifies the relative position of the two sites which the joint connects. The `connect` statement specifies that the joint connects one site *to* another, and this defines the direction of the joint and the joint's displacement. The first site in the connect statement is sometimes called the "from" site; the second site is called the "to" site. The joint's displacement transform defines the global placement of the "to" site relative to the global placement of the "from" site[1]. The directionality of the joint is defined in terms of how the joint would normally behave in the figure, when the figure is rooted in its normal place. By "normal" behavior, we mean that when you interactively adjust a joint, the "from" side remains stationary and the "to" side moves. This directionality remains fixed even if the figure is re-rooted and the actual figure hierarchy is changed.

For example, in the human figure model, the shoulder is defined to connect the clavicle *to* the upper arm. This definition is convenient since we normally expect that when we change the transformation at the shoulder,the arm will move and the upper torso will remain fixed. This will not be the case, however, if the figure is rooted through his hand, possibly hanging from a rope or floating in zero gravity. In this case, we still define the transformation across the joint in the direction "from" the clavicle "to" the arm.

A.3.5.1 Degrees of Freedom

Joints may have specific degrees of freedom which restrict the transformation across the joint. The type of the joint is specified by the `type` field, whose value is an arbitrary expression composed of primitive rotation and translation operators. The rotation operator is `R`, and the translation operator is `T`. Each rotation and translation specifies an axis. The axis must be a coordinate axis of unit length: $(1,0,0)$, $(-1,0,0)$, $(0,1,0)$, $(0,-1,0)$, $(0,0,1)$, or $(0,0,-1)$.

The current angle associated with the axis comprises the *displacement* of the joint and is not part of the type. A simple joint may be defined as:

```
joint elbow {
    connect lower_arm.proximal to lower_arm.distal;
    type = R(1,0,0);
    displacement = (90deg);
    ulimit = (180deg);
    llimit = (0deg);
}
```

This joint rotates only around the x axis. The displacement field specifies that the transformation is a 90° rotation around the x axis.

[1] The global placement of the "from" site is determined in turn by its location relative to its segment's base coordinate frame. This in turn is determined by other site locations and joint displacements, up the figure hierarchy to the root of the figure, where the global placement is fixed.

Joints may have up to three degrees of freedom by multiplying primitive operators. In this case, the displacement field has the same number of arguments as the type expression has primitive elements. The complete transformation at the joint is the product of each primitive operator instantiated with the appropriate angle. As in the case of homogeneous transforms, the operators should be interpreted right to left as local transformations, i.e. with respect to the local, or current, transform. Alternatively, the displacement at the joint may interpreted from left to right as primitive transformations with respect to the parent coordinate frame, that is, the site on the "from" side of the joint.

For example, the transformation at the joint defined by:

```
joint luceille_ball_joint {
    connect torso.shoulder to arm.base;
    type = R(0,0,1) * R(0,1,0) * R(1,0,0);
    displacement = (90deg,45deg,30deg);
    ulimit = (180deg,90deg,60deg);
    llimit = (0deg,0deg);
}
```

may be interpreted as a rotation of 30° around the x axis of site `torso.shoulder`, followed by a rotation of 45° around the rotated x axis, followed again by a rotation of 90° around the *rotated* y axis. It may alternatively be interpreted as a rotation of 90° around the x axis of the site `torso.shoulder`, followed by a rotation of 45° around the y axis of the same frame, followed by a rotation of 30° around the x axis of the same frame as well.

A.3.6 Constraint Declarations

Constraints specify desired geometric relationships. The parameters of constraints are described in Chapter 10. The syntax of the constraint declaration in the peabody language has an assignment statement for each of the constraint's properties. There can be no blocks within a constraint block.

A constraint has the following assignment fields:

type The relationship type. The expression can be either a single string giving the position or orientation type if there is only one, or its is a 3-vector, with the first element a string describing the orientation relationship type, the second a string describing the position relationship type, and the third element a number giving the position/orientation weight.

end The end effector. The value may be a site or a node. It must be typecast to determine which type it is.

goal The goal. The value may be a site, node, face, or a matrix. A matrix type specifies a hold constraint. Otherwise, it must be typecast to determine which type it is.

startjoint The starting joint. If this field is absent, then the constraint is a rooting constraint.

weight The constraint weight.

A.3.7 Surface Attributes

Each polygon in the peabody environment has a surface attribute associated with it, and the surface attributes may be specified in the peabody file. Surface attributes are defined in a block structured manner similar to the other peabody constructs:

```
attribute brown {
    diffuse = (0.48,0.26,0.00);
}
```

Surface attributes have the following assignment fields

ambient This is a triplet of real numbers describing the ambient color of the surface in RGB coordinates.

diffuse This is a triplet of real numbers describing the diffuse color of the surface in RGB coordinates.

specular This is a triplet of real numbers describing the specular color of the surface in RGB coordinates.

glossiness This is an integer exponent describing the glossiness.

Alternatively, you may specify the ambient and diffuse parameters as scalar values and give the attribute a color with the `rgb` field. This makes the ambient and diffuse parameters different intensities of the same color. This is the most common way of describing attribute parameters.

The way in which the surface attribute parameters affect the shading of the surface is described in Section 12.

A.3.8 Light Source Declarations

Light sources are necessary for rendering and are an integral part of a computer graphics and animation environment. Light sources are represented in peabody as special types of segments. A light declaration begins with the keyword `light`. Usually, a light source will consist of a figure with a single segment, but it is possible to give light source properties to any segment on any figure. The origin of the light is the origin of the light segment, and the light is emitted equally in all directions. Lights are not directed. In the peabody grammar, lights are interchangeable with segments:

```
figure foo {
    light bar {
        site base->location = trans(0cm,0cm,0cm);
        color = (1,1,1);
    }
}
```

Lights have the following assignment assignment fields:

ambient This is a triplet of real numbers describing the ambient color of the light in RGB coordinates.

color This is a triplet of real numbers describing the color of the light's emission in RGB coordinates.

The way in which the light source parameters affect the shading of a surface is described in Section 12.

A.3.9 Peabody Block Structure

The block structure of the peabody language facilitates prefixing constructs with their "parent" construct by effectively prefixing everything within the block with the appropriate construct name. Thus, within a figure declaration, all segment names are taken relative to the figure, so there is no need to prefix them. Within a segment declaration, the site names are taken relative to the segment. The joint declaration references sites, which must always be qualified at least to the segment level.

The block structure exists primarily for convenience, since all constructs may occur in any arrangement provided they are fully qualified. For example, a segment block normally belongs inside a figure block, but it may occur outside, provided the segment name is prefixed with the proper figure name.

The curly braces serve to prefix each assignment statement in a construct with the proper construct name and type. They may be replaced by the "arrow" notation:

```
segment cube->psurf = "cube.pss";
```

This short form may be used for any assignment field in the construct.

The traditional block structure resembles a "definition," but actually each construct is defined whenever it is first used. There is no formal distinction between definitions and references. Any subsequent references to the construct refer to the original rather than creating a new one. This allows the same file to be read multiple times without side effect. The effect of reading a peabody file multiple times is to reset the values of the assignment fields rather than create new constructs. A common use of this is for separating joint displacements from figure definitions.

This also allows for very terse definitions. For example,

```
site fred.arm.base->location = trans(0,0,0);
```

is a valid figure definition.

As *Jack* reads a peabody file, it sets the given construct field regardless of its previous value. An exception to this rule applies to psurfs. If a psurf for the segment already exists and it was read from a file of the same name, then the psurf is not re-read. This allows figure definitions which include psurf specifications to be re-read without the overhead of re-reading the psurfs.

A.3.9.1 Figure Files

Sometimes it is convenient to create a *definition* for a certain type of figure, that is, a template which can be used to create different instances of the figure.

A single environment file cannot be used to create different instantiations of the same figure definition, since the names are associated directly with the figures, and figures must have unique names. Peabody allows you to define figures in special *figure files*. Different figures may be instantiated from a figure file with different names.

A figure file is simply a file containing the definition of a single figure, without the identifier following the `figure` keyword. For example, consider a file called `thing.fig`:

```
figure {
    segment x {
        psurf = "cube.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(100,100,100);
    }
    segment y {
        psurf = "pyramid.pss";
        site base->location = trans(0,0,0);
    }
    joint elbow {
        connect x.p to y.base;
    }
}
```

Now, another file called, say `things.env`, could contain:

```
figure ["thing.fig"] a;
```

The syntax of the figure instantiation specifies the name of the figure file, `thing.fig`, in square brackets, followed by the name of the instantiated figure, `a`. Note that the name of the file must be enclosed in double quotes.

The file `things.env` may include an additional "thing":

```
figure ["thing.fig"] a;
figure ["thing.fig"] b;
```

A.3.9.2 Figure Parameters

Figures may have parameters, much the same as function parameters in a programming language. This makes it possible to generalize a figure template to be instantiated with different sizes, types, etc. For example, the above figure may have been defined as:

```
figure (height,width) {
    segment x {
        psurf = "x.pss";
        site base->location = trans(0,0,0);
        site p->location = trans(width,height,0);
    }
    segment y {
        psurf = "y.pss";
```

```
        site base->location = trans(0,0,0);
    }
    joint elbow {
        connect x.p to y.base;
    }
}
```

and the instantiation the figure would look like:

```
figure ["thing.fig"] (50cm,100cm) a;
figure ["thing.fig"] (40cm,200cm) b;
```

A.3.10 Postures and Posture References

△

A posture block is declared within a figure block. It's purpose is to capture a posture (joint angles, figure location, root, and constraints and behaviors) for a particular figure. There are two forms of a posture declaration inside a figure block. The first one is called a *posture reference*, and uses the keyword `postureref`, and looks like:

```
figure human {
    postureref ["standing.post"] (10cm) standup;
    postureref ["sitting.post"] sitdown;
}
```

The above declaration adds two named postures to the figure *human*, and they are named *standup* and *sitdown*. The important point to know here is that a `postureref` declaration will **not** read the posture files (in this case, `.post`) when the postureref is read. It only adds the named posture (along with filename and arguments) to the list of postures attached to the figure. This is important, since the posture definition file (`.post` file) may contain references to peabody objects that do not yet exist in the environment. The posture file is read when a *Jack* command requests the posture to be instantiated. This also allows `postureref` declarations to be stored in figure definition files (`.fig` files).

The other form of a posture declaration is called a *posture instantiation* and looks just like above, except `postureref` is replaced with `posture`. It would look like:

```
figure human {
    posture ["standing.post"] (10cm) standup;
    posture ["sitting.post"] sitdown;
}
```

The difference here is that when the declaration is read, the posture file is read, and therefore the posture is instantiated (the figure is moved to the corresponding posture). In the above example, the figure would be moved to the *standup* posture, then the *sitdown* posture.

Posture definition files can be parameterized just like figure files and motion group files (see above and below). In the example above, the posture file `standing.post` takes one parameter, a length measurement. The file `standing.post` could look like:

```
posture (length) {
    root = lower_torso.distal;
    joint left_shoulder->displacement = ...
    ...
    location = trans(length, 0cm, 0cm);
}
```

A posture file must always start with `posture`, not `postureref`. Inside a posture block, you may have joint displacements, site locations, constraints, behaviors, root specifications, etc (any field or block which may appear in a figure block may appear in a posture block). Note that all segment/site/joint names are *relative*, they are not prefixed with the figure name. This allows posture files to be *shared* between similarly structured figures (i.e. the human).

A.3.11 Motion and Motion Group Declarations

△

The declaration of a motion begins with the keyword `motion`, followed by a name for the motion. The naming convention is the same as that for segments. A motion should not be declared inside any other peabody construct, except a `motiongroup` or at the top level by itself (in which case it belongs to the `default` motion group. A sample motion in peabody format looks like:

```
motion hand {
    figure = (figure)human5;
    type = "hand control";
    starttime = 0sec;
    duration = 2sec;
    off = 1;
    velocitycontrol = "ease in/ease out";
    data = ("right","site",(site)hand.paths.point,"waist");
}
```

Every motion type has a set of common fields (figure, type, starttime, duration, off, and velocitycontrol). Additionally, each motion type defines a field called data which is simply a vector of values. This vector is generated by the motion to save whatever parameters it needs. It is also parsed by the motion when being read from a peabody file. The only restriction on the values is that it must be a valid parse-able peabody value. The assignment fields for a motion must appear in the order that they are shown above. The assignment fields for a `motion` are:

figure This is a figure reference to the figure to which this motion is applied. Since this is a figure reference, you must precede a figure name with the `(figure)` casting operator. If this is a peabody variable, it must be a figure reference. (See the `motiongroup` below).

type The type field is a string describing the type of this motion. The type tells peabody how to interpret the data field below. Please see the table below for the values of type, and the corresponding format of the data field.

starttime This is the time, in seconds, when the motion will start. This may be ant valid peabody expression which resolves to a number.

duration An expression, evaluating to the duration, in seconds, of the motion.

off This field is optional, and if 1, means the motion is actually turned off (it won't execute).

velocitycontrol For motions which interpolate between a beginning and ending value, this is the method for controlling the velocity function of the interpolation. Valid values are: `"constant", "accelerate", "decelerate", "ease in/ease out"`.

data The data field is a vector of values representing the information needed by each motion type. It is different for each type. Below is a few examples from the simpler motion types:

joint $(n, jointname_1, (displacement_1) \ldots jointname_n, (displacement_n))$

n is the number of joints involved. For each joint involved, the $jointname_i$ is listed, followed by a vector giving the goal joint displacement.

figure $(xform)$

The $xform$ is the global transform for the goal of the figure motion.

camera $(xform, window_name, vrpd)$

The $xform$ is the global transform for the camera, the $window_name$ is the name of the peabody window, and $vrpd$ is the view reference point depth for the camera.

light (*segmentp*, $(s_R, s_G, s_B), (e_R, e_G, e_B)$)

The *segmentp* is a segment pointer (to the light source), followed by the starting rgb color vector (s), and the ending color vector (e). If s is $(-1, -1, -1)$, the starting color is just the current color of the light source (when the motion starts).

item[path] (*segmentp*, *traversal*)

The *segmentp* is a segment pointer to the path, followed by *traversal*, which can be either `''forward''` or `''reverse''`.

item[figure path] (*segmentp*, *traversal*, *figurep*)

Same as above, except *figurep* points to the figure which moves along the path.

command (*preactionJCL*, *applyJCL*, *postactionJCL*)

These are three JCL command strings. The *preactionJCL* is executed on the first frame of the motion, the *applyJCL* is executed on each frame of the motion, and the *postactionJCL* is executed on the final frame of the motion. If the JCL has double-quotes, they must be appropriately escaped (i.e. " becomes
").

Each motion can also be a member of a motion group. Motion groups can also be parameterized similar to the way figure files can be parameterized (See Section A.3.9.1). An example motion group file:

```
motiongroup (fig, start, durate) {
    motion arm4_arm1 {
        figure = fig;
        type = "joint";
        starttime = start + (0 * durate);
        duration = 0.67 * durate;
        velocitycontrol = "constant";
        data = (3,jointref(fig,''joint4''),(0.00deg,0.00deg)
        ,jointref(fig,''joint3''),(0.00deg,0.00deg)
        ,jointref(fig,''joint2''),(0.00deg,0.00deg)
        );
        }
    motion chain {
        figure = fig;
        type = "figure";
        starttime = start + (0.33 * durate);
        duration = 0.67 * durate;
        velocitycontrol = "constant";
        data = (trans(5.62cm,0.00cm,-202.27cm));
        }
}
```

If this motion group were in a file call `move.mgp`, then a valid peabody reference to create this motion would be:

```
motiongroup ["move.mgp"] ((figure)chain, 0sec, 1.50sec) move;
```

The motiongroup structure is useful for grouping related motions together, and the intention of it is also to be used for creating composite motion templates. But there is a major problem ... The problem is that peabody name references within the data field of motions need to be *relative* to the figure of the motion or relative to some parameter of the motiongroup.

A solution to this problem is to use the peabody built-in functions (siteref), `segmentref`, and `jointref`. These function takes two arguments, the first can either be a figure name (string) or a figure reference, and the second is a *relative* site/segment/joint name (i.e. of the form `segment.site`, `segment`, or `joint` respectively). These function will return a reference pointer to the actual object (site, segment or joint).

A.3.12 The Include Statement

The `include` statement nests a peabody file in another file, similar to the `#include` preprocessor control statement in the C programming language. Its format is simple:

```
include "file.env";
```

The effect is identical to inserting the entire contents of the file into the original file at the location of the include statement. Files may be nested to a level of 8.

A.4 The Syntax of Psurf Files

Psurfs may be described syntactically in text files. A psurf file is a textual representation for the nodes and faces of the psurf. By convention, these files have the suffix `.pss`. The file lists a set of nodes and faces.

```
{ a cube:  8 nodes, 6 faces }
0.00    0.00    0.00
0.00    100.00  0.00
100.00  100.00  0.00
100.00  0.00    0.00
0.00    0.00    100.00
0.00    100.00  100.00
100.00  100.00  100.00
100.00  0.00    100.00
;; { end of nodes }
1 2 3 4; { back }
1 4 8 5; { bottom }
3 7 8 4; { right side }
1 5 6 2; { left side }
5 8 7 6; { front }
2 6 7 3; { top }
;;
```

Figure A.5: `An Example psurf`

A psurf file begins with a list of nodes, which are specified as triplets of real numbers. There may be an optional comma between the triplets. The numbers may contain decimal points, but the decimals are not necessary. No leading 0 is required for fractions less than 1.0. The nodes are numbered implicitly starting at 1. The node table is terminated with two semicolons.

Following the nodes are the faces, which are lists of indices into the nodes. Each list specifies the vertices of the face, and is terminated with a semicolon. The faces are terminated by an empty vertex list, i.e. two adjacent semicolons. There is a predefined limit of 256 vertices in each face. The vertices should be in counter-clockwise order. This is discussed in greater detail in Section A.4.5.

Comments may appear anywhere in the file and are delimited by curly braces. The indices listed in the psurf file all start at 1 for historical reasons. A single psurf file may contain several individual "psurfs." In other words, is is legal to concatenate several psurf files into one, which will make a single psurf out of several disconnected components.

Psurfs may be in any general scale, but it is best to make the coordinates correspond to centimeters. *Jack* assumes that all objects are scaled in centimeters, and it initializes the view so that objects in the same general scale as human bodies are displayed conveniently on the screen. You can change this scale if necessary, but it is still best to keep all object in the same basic range of sizes.