How to use Juno.

Juno is a Cedar program that you can use to draw pictures. Juno is interactive, so that you can see the picture on your screen as you draw it, and point into it with the mouse. Juno is also programmable, so if the basic system doesn't do what you want you can write a program in the Juno language that does, and invoke your program interactively just as if it were a built-in command. Juno is also ''constraint-oriented''. This means that you specify your image implicitly by declaring the geometric relations within it, instead of explicitly programming the computations of the locations in the image.

Here are some reasons for learning to use Juno:

> You can use Juno to draw certain kinds of pictures that would be di cult to draw with SIL or Gri en.
>
> You can sharpen up your skill in geometry.
>
> You can learn something of the real nature of constraint-based systems by getting hands-on experience with a esty one.
>
> You can learn something about programming languages by using one simple enough that you draw its programs instead of typing them. For example, to write a program to draw a circle, you don't type a word: you draw a circle on the display, then direct Juno to extract the underlying program.
>
> You can impress visitors to PARC with ashy demos.

But you should be warned that Juno is a tricky program. It is not complicated, but it takes a long time to master. So you should start early.

Most people are eager to try a system rather than read about it, so we will begin with instructions for test-driving Juno.

Getting started. Seat yourself comfortably at a machine that is running Cedar 5. Prop these instructions up where you can read them; you'll need two hands to operate Juno. Fasten your seat belt.

Use the DF tool to bringover /indigo/juno/juno.df. Then type "Juno" to the command tool, and wait patiently for about twenty seconds, while Juno starts itself and opens two viewers. The rst is a command window called Juno Buttons and implemented with Mike Spreitzer's Viewrec package. The second displays an image and is called "Juno Image". The Image viewer needs at least two-thirds of a column to display itself; usually it shares a whole column with the Buttons viewer.

Learn to pick up the cursors that are lined up at the left of the Juno Image viewer by moving the mouse to the left until the cursor becomes a right-pointing arrow in the left margin, then moving

the mouse up or down until the arrow points at the cursor you want, then moving the mouse to the right, as though you were spearing the desired cursor with the arrow.

Now learn to draw a triangle by following the instructions in gure 1. Here is how to interpret the instructions. The picture of the pencil icon means to pick up the pencil cursor. A upper- case letter means to click with the left mouse button in the Juno Image viewer at the approximate position of the point with that letter as its name. A lower case letter means to click with the middle mouse button at or near the point with that letter as its name. means to hit the escape key.

In other words, to draw a triangle with vertices at positions p, q, and r, starting from a blank screen, pick up the pencil, left click at p, left click at q, hit escape, left click at r, hit escape, middle click at p, and hit escape. Then put down the pencil.

If you make a mistake, click the StartOver button in the Juno Buttons viewer. (The button stays greyed after it is nished starting over. Why? Because that's Viewrec's way. Sometimes, due to a repainting bug, Startover fails to clear all the highlights from the screen; they will dissapear next time the screen is repainted. If you destroy the Juno viewer, can't run it again without rolling back.)

Now learn to use the T-squares to line things up horizontally and vertically, by following the instructions in gure 2 for drawing a rectangle. The instructions are intepreted just like the instructions in gure 1. You rst draw an approximate rectangle with the pencil, not worrying about getting it exactly right. Then you use the horizontal T-square to atten the top and bottom, and the vertical T-square to plumb the sides.

In case you're getting tired of straight lines, the next exercise has a curve in it. Follow the instructions in gure 3 to draw a "Bezier curve" determined by four points. Notice that the curve lies entirely in the quadrilateral determined by the four points; this is emphasized in the gure by drawing the edges of the quadrilateral as well as the arc. The Bezier curve `arc(p,q,r,s)` starts at p and ends at s, but it does not go through q or r: these two parameters control the direction of the curve at the endpoints. If you walked the curve you would start at p facing q and end at s with r directly behind you.

Try moving the four points to see how the shape of the curve changes. Figure 3 shows how to move points: pick up the move arrow, middle- click the point you want to move, and left-click the place you want to move it.

It is instructive to draw `arc(p,q,s,r)` in the same quadrilateral; this curve is more S-shaped.

You may be wondering how Juno chooses which points to move when it is solving the constraints. The answer is: according to the whim of its solver. But there is a way to restrict the solver's choice. In any Juno image, some points are *frozen*. The solver will never move these points. To nd out which points in the image are frozen, pick up the Snowman cursor. The points that are frozen will all be highlighted. Clicking a point with the middle button while holding the Snowman cursor will ip its state from frozen to unfrozen and back again.

To practice with the snowman, try drawing the symmetrical horseshoe shape shown in gure 4. The shape consists of two arcs, one on the left and one on the right. By following the directions with

the T-square and compass, you can constrain the left and right sides to be symmetrical. Then you can use the move arrow to ne- tune the picture to the exact shape that you want. (For practice, try to match the shape in the gure.) But you will nd that the whim of the Juno solver is hard to live with, and that your ne adjustments to your horseshoe don't work because the solver moves the wrong points. Snowman to the rescue! Try freezing the points p, v, r, s, and t, and adjusting the position of u, as suggested in the instructions. Other useful techniques are: freeze the positions of q, u, r, t, and s, and adjust the position of v; or freeze the positions of p, v, q, u, and adjust the position of t.

Now that you have the feel of Juno, you should be ready for a more thorough exposition of its principles.

What an image is. A Juno image is determined by a set of points, a set of constraints on the positions of these points, and a set of painting commands parametrized by the points. Each time the image is printed or displayed, the constraints are allowed to act upon the points, pushing them to a nearby solution, and then the painting commands are executed upon an initially blank canvas. The internal representation of an image is a symbolic expression in which the points are identi ers, the constraints are predicate symbols applied to the identi ers, and the painting commands are procedures taking xed numbers of point parameters.

Figure 14 shows a typical Juno image, with four points, labelled a, b, c, and d. These points are constrained by three constraints. The rst constraint is that (a,c) is parallel to (c,d), that is, that acdb is a rhombus. The second constraint is that (a,c) is also congruent to (c,d). (Two gures are congruent if they coincide when properly superimposed. In particular, line segments are congruent if they have the same length. Thus the second constraint is that segment (a,c) equals segment (b,d) in length.) A rhombus whose opposite sides are equal is a parallelogram, so acdb is a parallelogram. The third constraint is that the diagonals of this parallelgram are equal; a parallelgram whose diagonals are equal is a rectangle. (It happens that in the image of gure 14, the rectangle is a square, but this is a coincidence, not enforced by the constraints.)

There are four commands in the image of gure 14. The rst command, `edge(b,d)`, draws a line segment between the points b and d; the second command, `circle(c,d)`, draws a circle that has center c and contains d. The segment and circle are present in the image. The third command, `yellowSubmarine(a,c)`, draws a yellow Trident submarine with snout a and tail c; the submarine is present in the image. The third command, `arc(c,d,a,b)`, draws a curved line from c to b; the points d and a control the shape of the curve.

The commands `arc` and `edge` are primitive; the commands `yellowSubmarine` and `circle` are compound commands that are executed as a sequence of primitive commands.

It is important to understand that only points can be constrained; segments, circles, submarines, etc. are not constrainable objects, but side e ects created by the execution of commands parametrized by points. (The point labels and the black dots in gure 14 were added for reference purposes; they are not really part of the image described by the text in the gure.)

Every constraint of a Juno image has one of the forms

`hor(p,q)`	The segment from p to q is horizontal
`ver(p,q)`	The segment from p to q is vertical

`(p,q)cong(r,s)` The segment from `p` to `q` has the same length as the segment from `r` to `s`
`(p,q)para(r,s)` The segment from `p` to `q` is parallel to the segment from `r` to `s`

Other constraints, for example that two segments be perpendicular, must be replaced by equivalent combinations of these four. Thus by using Juno, you will sharpen your geometric skills.

The complete class of Juno painting commands will be described later. Two important elementary painting commands are:

`edge(p,q)` Paint a straight line segment from `p` to `q`
`arc(p,q,r,s)` Paint the cubic polynomial curve determined by `p`, `q`, `r`, and `s` and Bezier's rule.

Operating on images. You can operate on the image by adding or deleting points, constraints, or painting commands, or by changing the locations of points. Altogether there are twelve operations, each of which is invoked by its own cursor. Figure 5 shows the twelve cursors, their names, and their functions. You pick up a cursor by spearing it cleanly from the left.

To create a new point in the image, point with the mouse to the place where you want it and push the left button. This creates the new point and simultaneously enqueues the point as a parameter for a later operation, in a data structure called the *argument queue*.

The points on the argument queue are highlighted with crosses on the screen. A point that is in the queue twice will be highlighted both with a cross and an ex. (Sometimes, when Juno gets confused, it highlights a point twice with the same mark; then the highlights cancel one another and it looks like the point is not in the argument queue at all. When this happens, you just have to remember which points you clicked.)

To put an existing point on the argument queue, point in its vicinity and push the middle button. If the highlighting reveals that you have pointed inaccurately, correct before lifting the button.

You can think of Juno as a simple stack machine, which you control by pushing arguments and calling for operations. To apply an operation to the existing argument queue, hit the escape key. The operation is determined by the current cursor and the depth of the argument queue. Figure 6 shows how to add the elementary constraints and commands. The e ect of the escape key is only to add the constraint or command to the image, not to redisplay. Hitting escape a second time will cause a redisplay.

Adding constraints, edges, and arcs. After a cursor does its thing with the current arguments, it pops some of them from the queue. But not all of them. It leaves some as arguments for future operations. This is because graphics operations often follow a path of points; each point of the path is the last argument of some operation and the rst argument of the following one. The last column of gure 6 shows how many arguments are left in the queue by the constraint cursors and the pencil.

For example, to enter the constraints `hor(p,q)`, `hor(q,r)`, and `hor(r,s)` into the image, pick up the horizontal T-square and enter

`p q r s`

where is used to mean hitting the escape key. Because `q` is left on the stack by the rst , the second adds the constraint between `q` and `r`. To enter `arc(a,b,c,d)`, `edge(d,e)`, and `arc(e,f,g,a)`,

pick up the pencil and enter

```
abcd e fga .
```

The argument queue never becomes deeper than the maximum allowed by the current cursor. The maximum depths are 2 for the T-squares and 4 for the pencil, compass, and parallel bars. For example, to enter `hor(p,q)` and `hor(r,s)`, pick up the horizontal cursor and enter

```
pq rs .
```

Because the depth of the queue is limited to 2, point `q` will be removed from the queue when `s` is added, leaving just `r` and `s` on the queue for the second operation.

In order to facilitate the transition from a pencil path to a second, disjoint pencil path starting in a straightedge, the e ect of hitting escape with the pencil when there are three arguments on the queue is to draw a straight edge, as shown in gure 6. For example, to enter `edge(p,q)` and `edge(r,s)`, pick up the pencil and enter

```
pq rs .
```

Point `q` is ignored by the second operation.

Moving and freezing points. After the constraints and commands are entered into an image, it is usually necessary to move the points around to get the image right. A point that you move is e ectively a parameter or independent variable; a point that the solver moves in response to your motion of the parameters is e ectively a dependent variable. It is important to keep this distinction in mind: otherwise you will nd yourself moving points at random, frustrated by the whim of the solver. The great value of constraint- based methods is that they allow you to easily change which variables are independent and which variables are dependent. But clear thinking is required to exploit this capability. To the extent that constraints take much of the tedium out of drawing pictures with a computer, they leave to the user undiluted di culties.

The Move arrow is used to move points or groups of points. Groups will be discussed later. To move a single point, put the source and destination on the argument queue and hit escape. The e ect is shown in gure 7. Usually `p` is clicked with the middle button and `q` with the left button; in this case the step ‘‘identify `p` and `q`’’ simply absorbs `q`, which has no further role. But you can click `p` and `q` both with the middle button if you want to weld two existing points into one. If `p` starts frozen, it remains frozen; if it starts thawed, it remains thawed; but in any case it will be frozen at `q`’s position during the solve step of the move operation.

The Move operation can also be invoked by hitting tab instead of escape; this suppresses the solve step in the redisplay, so that just the one point is moved. You can sometimes use this feature to untangle a drawing that the solver has messed up.

No simple rules can explain the behavior of Juno’s constraint solver, which uses Newton- Raphson iteration. Those who are interested in numerical methods will nd it amusing to construct images that illustrate the idiosyncracies of this method. I have two suggestions for avoiding the pitfalls of Newton- Raphson. First, parallel constraints involving short segments can lead to numerical instability. Thus if `a` and `b` are close together and far from `c`, and you want to constrain them all to be collinear, enter `(a,c) para (b,c)` rather than `(a,b) para (b,c)`. Second, in a few circumstances,

notably the drawing of regular polygons with ve or more sides, the existence of redundant constraints leads to numerical instability. In such cases, take care to enter the minimum number of constraints that are required to specify the design.

At this point you may want to try your hand at drawing some of the diagrams in gure 8. Most beginners nd these diagrams quite challenging, so don't be discouraged if you can't draw them the rst few times you play with Juno.

Strings and Extensions. If you want more than edges and arcs, you can use the X cursor or the typewriter to add other painting commands to images.

If `p` is a point, `s` is a string of characters, and `f` is a font indicator, then `type(p,s,f)` is a Juno painting command. When the image is printed or displayed, this painting command will paint the string `s` at the position `p`, using the font `f`. To add such a command to the current image, pick up the typewriter cursor, click `p` with either the left or middle button in the usual way, and then type the string `s`. End with escape, or with newline if you want to type another string directly below the rst. By default the font is ten-point helvetica; to change it, click the ChangeFont button in the Buttons viewer, ll in the elds, and push the "Do ChangeFont" button. Don't include the face information in the font name; for example, don't name a font TimesRomanb, name it TimesRoman and set the bold eld to TRUE.

If Juno does not have the screen rastors for the font that you select, it will display the string in Tioga's default font, but it will put your requested font into the press le. So, for example, you can ask for TimesRoman in a huge size; the letters will be small on the screen, but correct in the printed version. The fonts that Juno can display correctly on the screen are those in the directory /indigo/tioga/strikefonts.

Sad news: You will nd that it is possible to select the right endpoint of a string, for example to align it vertically with the right endpoint of another string. This is inconsistent with the explanation in the last paragraph but one, where a string was a painting command parametrized by a single point. The previous paragraph is the truth; the selectability of right end-points of strings is an error and it should not be relied upon. Juno doesn't know about the printer widths of fonts, only about screen widths, so the image on the screen is not a reliable facsimile of the image that will be printed.

The X cursor is used to call a procedure. So before you use it, you have to get hold of some procedures. Juno lets you have only one le of procedures active at a time, and the "Algebra" button lets you switch les. To experiment with the X cursor, push the Algebra button, ll in the leName eld with /indigo/juno/1/basic.juno, and then push the "Do Algebra" button. After a few moments a viewer will appear containing several Juno procedures. Don't try to do anything while the Algebra button is working, because it uses the Tioga selection to process the speci ed le. The viewer is an ordinary Tioga text viewer with exactly two levels of node structure; FirstLevelOnly will display all the procedure names, any MoreLevels will display the whole le, containing name-de nition pairs.

Choose one of the procedures in basic.juno and select its name by pointing at it with the mouse and clicking with the middle button. Then click the LoadX button. Now, whenever you hit escape while holding the X cursor, you add a new command to your image; the command is an application of the procedure whose name was selected when you hit the LoadX button, and the arguments of

the application are the contents of the argument queue when you hit escape. Thus the number of arguments that the X cursor takes is the number of parameters of the procedure with which it is associated.

Another le of Juno procedures that you can look around in is /ivy/gnelson/JunoManual.juno, which contains the procedures used to draw the gures and tables for this manual.

Operations on groups. So far we have described operations on images that take xed numbers of point parameters. Next we will describe operations on groups of points, constraints, and commands.

A group is speci ed by holding down the right mouse button and rolling the mouse around the points that you want to include in the group. When you push the right button, the mouse begins to leave a visible track on the screen; the track ends when you lift the button. A point is included in the group i the track of the mouse winds around it a non-zero number of times. This is called balloon selection, because the track left by the winding mouse looks like a balloon. A constraint or command is considered to be in the group if all its parameters are in it. Notice that the path of the mouse may wind around an arc without winding around the four parameter points of the arc; in this case the arc is not in the group.

The simplest group operation is performed by the eraser. To try out the eraser, draw a few edges and arcs, then pick up the eraser, draw a balloon around some of your points, and hit escape. Here's what will happen: Every constraint or command in the group will be deleted from the image. Also, every point that is in the group and is not connected to any point outside the group will be deleted, where two points are considered connected if they are both parameters of some constraint or command.

The move arrow, copy arrow, and Y cursor operate on groups together with some number of points of the group designated as sources for the coming operation. A group with sources is de ned entirely with the right mouse button, in two steps. First the group is de ned by balloon selection, as with the eraser. Then one, two, or three source points in the group are selected, by clicking them with the right button. (Because a balloon selection already exists when you click the source points, and because no operation takes two groups, Juno assumes these right button clicks are for indicating source points, rather than for another balloon selection.)

Here is how to move a group, either translating, rotating, or sheering it along the way: Pick up the move cursor, de ne the group (by balloon selection) and the sources (by right clicking), and then specify one target point for each source, using the left or middle buttons in the usual way. Finally, hit escape. Juno will compute a linear transformation that carries each source point into the corresponding target point and apply the transformation to each point in the group. It will also identify the source points with the target points and redisplay the image. During the solve step of this redisplay operation, Juno will not let the source points budge from their new positions.

With one source and one target, the transformation will be a translation. With two sources and two targets, the transformation will be the composition of a translation, a rotation, and a change of scale. With three points, the transformation will be a general linear transformation. Note that in order to obtain a re ection, it is necessary to use three sources and targets.

The copy cursor is similar to the move cursor. You specify a group to be copied, source points for

the group, and one target point for each source. When you hit escape, every point, command, and constraint in the group is copied, and then the copies are transformed as by the move cursor.

Programming Juno. A Juno command can be considered as acting upon an "abstract Juno machine". The state of this machine is determined by:

an indefinite number of named *point registers*, each of which contains a value representing a point in the Cartesian plane. There is one of these registers for each symbolic name.

a single anonymous *image*, to be imagined as a map from the Euclidean plane to the set of colors, but represented as a map from a fine discrete grid to the set of colors.

three *mode registers*, called the color register, width register, and endsType register, whose contents will be described later.

Mode Commands. The first commands we will look at are the mode change commands, which set the contents of a mode register, execute a subcommand, and then restore the original contents of the mode register. The syntax for these commands are:

Command ::= ... | *ModeChange* ...
ModeChange ::= *Color* paint *Command* | *EndsType* ends *Command*
| *Number* width *Command* | *Point*,*Point* width *Command*
EndsType ::= butt | square | round
Color ::= black | white | grey | red | blue | green | yellow | cyan | magenta
| darkred | darkblue | darkgreen | darkyellow | darkcyan | darkmagenta
| lightred | lightblue | lightgreen | lightyellow | lightcyan | lightmagenta

If c is a color and A is a command, then the command "c paint A" means "set the color register to c; execute A; then restore the old contents of the color register".

If e is an EndsType and A is a command, then the command "e ends A" means "set the endsType register to e; execute A; then restore the old contents of the endsType register".

If r is a real number and A is a command, then the command "r width A" means "set the width register to r; execute A; then restore the old contents of the width register".

If p and q are points and A is a command, then the command "p,q width A" means "set the width register to the distance from p to q; execute A; then restore the old contents of the width register".

Painting Commands. The elementary painting commands are to fill a path, stroke a path, draw a path, or print a string. The syntax for these commands is:

Command ::= fill *Path* | stroke *Path* | draw *Path*
| print(*String*, *Point*, *String*, *Number*, *Number*)

Path ::= *Patch* | *Patch*, *Path*

Patch ::= (*Point*, *Point*) | (*Point*, *Point*, *Point*, *Point*)

The last point of a patch should be the same as the first point of the patch that follows it in its path.

The effect of the command "fill p" is to change the color of every image point that p entwines to the contents of the color register. A path *entwines* a point if the winding number of the path with respect to the point is non-zero.

The effect of "stroke p" is to draw a stroke along the path p whose color is the contents of the color register, whose width is the contents of the width register, and whose ends are finished off in a style determined by the contents of the endsType register. This register contains one of the values butt, square, or round. For the exact meaning of drawing a stroke along a path, see the documentation for the Cedar procedure Graphics.DrawStroke, which Juno relies on to draw the stroke.

The command "draw p" is equivalent to "round ends black paint 1 width stroke p".

The command "print(s, p, f, x, n)", where s and f are strings, p is a point, x is a real number, and n is one of the integers 0, 1, 2, or 3, prints the string s at the position p, using the font whose family name is f, whose size in points is x, and whose face type is encoded in n according to the rule

0 = normal
1 = italic
2 = bold
3 = bold italic

For example, figure 9 shows the result of executing the command:

```
grey paint butt ends 15 width
stroke(a, b), (b, c);
stroke(d, e); stroke(e,f);
round ends stroke(g, h);
butt ends stroke(i, j);
square ends stroke(k, l);
fill(n, m, o, p), (p, q);
black paint 5 width
stroke(n, m, o, p), (p, n);
o, q width round ends stroke(q, q)
```

Compound commands. So much for the elementary commands. Among the compound commands, we can quickly dispense with composition: if A and B are commands, then so is their composition A;B, and it is executed by first executing A and then executing B. What is left is to describe the compound command that invokes the constraint solver. We will describe the general form of such commands, taken from Dijkstra's wp-calculus, before specializing to the case of the Juno abstract machine. In general a constraint-solving command has the form:

```
  if  List of variables to be introduced
  st  Constraint
then  Command
  fi
```

To execute such a command, an interpreter introduces the listed variables as locals, executes the given command, and then removes the local variables from the context. The initial values of the local variables must satisfy the given constraint. The choice of "if" and " " as keywords may seem strange; see CGN10 for an explanation.

For example, figure 10 shows Euclid's construction of an equilateral triangle. Here is a natural translation of Euclid's construction into the wp-calculus:

```
  if C, D
  st Circle(C) and Center(C) = A and On(B, C)
 and Circle(D) and Center(D) = B and On(A, D)
then   if E
       st Point(E) and On(E, C) and On(E, D)
     then Draw(A,E); Draw(B,E)
       fi
  fi
```

This translation is not a legal Juno program, because it has variables ranging over circles, and in Juno variables are only allowed to range over points. Here is a legal Juno translation:

```
  if E
  st (A, E) cong (A, B)
 and (B, E) cong (B, A)
then Draw(A,E); Draw(B,E)
  fi
```

So far we have ignored an important issue: what if the constraints do not have a unique solution for the local variables? For example, given the program above, which equilateral triangle will Juno draw, the one above the segment between a and b, or the one below? According to the formal semantics in CGN10, the interpreter is allowed to introduce any solution to the constraints if you care which solution, you should strengthen the constraints. But Juno's class of constraints cannot distinguish between the two solutions for the equilateral triangle.

I tried extending Juno's class of constraints to remedy this, but was unable to solve the extended class reliably.

So something else had to be done, namely: When introducing a local variable, you not only give a constraint on its initial value, you also give a hint where to find the solution to the constraint. Your hint is used as an initial value by the Newton-Raphson iteration algorithm that Juno uses to solve the constraints. Figure 11 illustrates the idea: The program hints that the solution for `E` is in the vicinity of the point whose coordinates are (1, 1) relative to `(A,B)`; that is, in the orthonormal coordinate system whose origen is at `A` and whose `x` unit vector goes from `A` to `B`. The interpreter uses the hint as the initial position of `E`, and then uses Newton-Raphson iteration to solve the constraints for `E`.

It is also possible to give a hint as an absolute offset from a single point, as in `x == (2, 3) rel (y)`, which hints that `x` should be 2 screen pixels to the right and 3 pixels above `y`. Finally, it is possible to give a hint in a sheered coordinate system; for example, `(2, 3) rel (a, b, c)` means the point with coordinates `(2,3)` in the coordinate system whose origen is at `a`, whose `x` unit vector goes from `a` to `b`, and whose `y` unit vector goes from `a` to `c`.

Assignment. The Juno language includes the assignment command

$$p_1,...p_n := q_1,...q_n$$

which sets the contents of the point registers named on the left to the contents of the point registers on the right. For example, `x, y := y, x` exchanges the contents of `x` and `y`. But the interactive Juno system doesn't understand about assignment, so if you use the X cursor to add commands to the Juno image that move their parameters via the assignment operation, the redisplay operation will lie to you, printing them in their old positions instead of their new ones. Since, in the absence of iteration, assignment is never really necessary, I advise you to avoid it.

The Y cursor. Figure 12 is a block diagram of the Juno system showing the role of the X and Y cursors. The X cursor introduces into the current image a call to some procedure that is defined in the currently active file of procedures. The Y cursor is a kind of inverse to the X cursor: it introduces into the file of procedures a new procedure whose execution will produce the current image, or a specified group in the current image.

To use the Y cursor, draw a balloon around the part of the image that you want to make into a procedure, select either one or two sources for the group, and then hit escape. Juno will insert into the current file of procedures a new procedure of the form

`X(`*Args*`) : if` *LocalList* `st` *Constraints* `then` *Command* `fi`

where

> *Args* is the list of sources that you specified. Thus the Y cursor only creates procedures with one or two parameters.

LocalList contains one point for each point in the group that you ballooned, exclusive of the sources, of course. The hint for the position of the point will be computed from the actual position of the point relative to the source or sources that you speci ed. If there is one source, the hint will be an absolute vector o set from the source. If there are two sources, the hint will be the point's coordinates in the coordinate system whose origen is the rst source and whose unit x-vector goes from the rst source to the second.

Command is the composition of all painting commands in the group, in the order that you entered them.

The Y cursor generates names for points by ennumerating the alphabet. It displays these names in the Juno Image viewer, so you can edit the procedure produced by the Y cursor to your speci cation. You always want to change its name, and you often want to promote some of its local variables into parameters and change some of its draw commands into stroke or ll commands. After making these parses you will need to reparse.

Parsing. The Juno command interpreter maintains a binding of procedure names to procedure bodies. This binding is initialized when you push the Algebra button on the Buttons viewer by parsing the contents of the le that you select. Thereafter, the binding is updated when you push the Parse button. You will need to do this every time you edit a procedure, or add a new procedure.

A le of Juno procedures is stored in a Tioga document with the following node structure: There is one rst-level node for every procedure in the le. If the procedure is short enough to t on one line, then the node for the procedure contains this line and has no subnodes. But most procedures don't t on a line; in this case, only the rst line of the procedure goes in the top-level node, and the rest of the procedure goes into the only child of the node. A procedure should end in a carriage-return character. If these rules are followed, and if the le has Juno.style applied to it, then the FirstLevelOnly button on the viewer will display the list of procedure names single-spaced, and the MoreLevels button will display the whole le, with a blank line between procedures.

When you hit the Parse button in the Buttons viewer, Juno examines each top-level node of the current procedure le to determine whether it has been edited since Juno last examined it. If so, Juno reparses the node, together with its rst child, if any. If the parse is successful, then Juno pretty-prints the procedure back into the node, and goes on to the next top-level node. If the parse is unsuccessful, then Juno moves the Tioga selection into the part of the viewer where it thinks the parse error occurred. Depending on the type of parse error, Juno may pretty-print the node fully or partially. Warning: if Juno thinks the parse error is at the very end of the node, it will position the Tioga selection to a point selection at the end of the node. As this is also what it does when the parse was successful, you have no warning of the error, and will not nd out about it until you call the procedure or try to load the X cursor with the procedure. This usually happens when you forget to type a closing " ", so be careful.

If, instead of using the Y cursor, you type a procedure directly into the current procedure le, begin by pushing the CommandForm button in the Buttons viewer, since it inserts a form for you to ll in with the correct node structure. Never use node structure yourself in a le of Juno procedures.

Juno.style calls for no indenting of subnodes, so its hard to see if the node structure gets messed up. To get a procedure de nition to the top level, select it and give the Tioga "unnest" command

until Tioga says "Can't do it". To get a procedure body to the correct level, select it and unnest until Tioga says "Can't do it"; then nest it once.

You may now want to try your hand at drawing some of the designs in gure 13.

Errors. Juno's philosophy about errors can be summarized: if you get an uncaught signal, you probably did something wrong. Because Juno is a research prototype, and because the Cedar system elds errors and allows you to continue from them, it is possible to use Juno even though it is so touchy. If you get an error, abort it and continue.

A common error is to call an unde ned procedure. This raises an appropriate error from the interpreter. Also, if in a Juno procedure you refer to a point that is not de ned in the current environment, you get an appropriate error. Almost any other error that you make for example, applying a procedure to the wrong number of arguments you get an inappropriate message, such as an error caused by attempting to dereference NIL.

History and related work. I conceived of Juno while I was living in Juneau, Alaska, inspired by the experience of designing a family of fonts with Knuth's Metafont system. I started coding Juno early in 1981. During the summer of 1982, Donna Auguste converted Juno to use the viewers window package, and added an early version of the X cursor. By the summer of 1983, I had added the Y cursor; this enormously increased the utility of the program. But by now Juno had outgrown the limitations of its original design, and further growth seemed to require complete reworking. I therefore started to work on Juno 2, but as progress has been slow, I am writing this document to encourage people to use Juno 1, which is described as "Juno" in this manual.

The closest related work to Juno that I know of is Ivan Sutherland's Sketchpad system. Alan Borning's Thinglab and Chris Van Wyck's IDEAL system are second cousins.

A nal thanks: to the many makers of Cedar, who built what Juno is built on; especially to the makers of Cedar graphics, who did everything right; and to Neil Wilhelm and David Dobkin for setting me straight about solving non-linear equations.

Appendix. The collected syntax of the Juno language:

ProcDef ::= *Proc* (*PointList*) : *Command*

Command ::= draw *Path* | stroke *Path* | fill *Path*
| *Assignment* | *ProcCall* | *ModeChange*
| print (*String* , *Point* , *String* , *Number* , *Number*)
| if *LocalList* st *Constraint* then *Command* fi

Assignment ::= *PointList* := *PointList*

ProcCall ::= *Proc* (*PointList*)

ModeChange ::= *Color* paint *Command* | *EndsType* ends *Command*
| *Number* width *Command* | *Point* , *Point* width *Command*

LocalList ::= *Local* | *Local* , *LocalList*

Local ::= *Point* == (*Number* , *Number*) rel *CoordinateSys*

CoordinateSys ::= (*Point*) | (*Point* , *Point*) | (*Point* , *Point* , *Point*)

Constraint ::= *AtomicConstraint* | *AtomicConstraint* and *Constraint*

AtomicConstraint ::= hor (*Point* , *Point*) | ver (*Point* , *Point*)
| (*Point* , *Point*) cong (*Point* , *Point*) | (*Point* , *Point*) para (*Point* , *Point*)

Color ::= black | white | grey | red | blue | green | yellow | cyan | magenta
| darkred | darkblue | darkgreen | darkyellow | darkcyan | darkmagenta
| lightred | lightblue | lightgreen | lightyellow | lightcyan | lightmagenta

EndsType ::= butt | square | round

Path ::= *Patch* | *Patch* , *Path*

Patch ::= (*Point* , *Point*) | (*Point* , *Point* , *Point* , *Point*)

PointList ::= *Point* | *Point* , *PointList*

Proc ::= Any sequence of letters and digits starting with a letter (case matters).

Point ::= Any sequence of letters and digits starting with a letter (case matters).

Number ::= A real constant without E notation not beginning with "." (e.g. 0.5 instead of .5)

String ::= A sequence of characters enclosed between double-quotes.

Juno allows you to type certain lexical tokens in more than one way: it does not distinguish between `->` and `then`; `:=` and `gets`; `|` and `st`. Most people find the latter forms faster to type.

The semicolon has higher binding power than `width`, `ends`, or `paint`, which are right associative among themselves.

Fig 1.

<==<JMF1.press<

Fig 2.

<==<JMF2.press<

Fig 3.

<==<JMF3.press<

Fig 4.

<==<JMF4.press<

Fig. 5.

<==<JMF5.press<

Fig. 6.

<==<JMF6.press<

Fig. 7.

<==<JMF7.press<

Fig. 8.

<==<JMF8.press<

Fig 9.

<==<JMF9.press<

Fig 10.

<==<JMF10.press<

Fig 11.

<==<JMF11.press<

Fig. 12.

<==<JMF12.press<

Fig. 13.

<==<JMF13.press<

Fig. 14.

<===<JMF14.press<