Tree properties:

* Each node/link is a d3 object, for which an SVG element is displayed
* Tooltips are divs that are positioned next to their node SVGs (divs to allow Latex compilation)
* Nodes
  + Store \_id field of parent and child nodes (along oriented links) in parentsIx and childrenIx arrays
* Links

tree\_of\_knowledge.html

* only “graph” template in the body
* includes all titles, buttons, MathJax call and graphSVG in a div “canvas”

tree\_of\_knowledge.js

* graph loading, re-loading, creating, deleting, etc. functionality
* different graphs are all stored in one DB on the server, with each link/node having a field “graph” that identifies which graph it belongs to.
* Current graph is stored in a Session var, which is accessed in the subscribe function in dbServer.js – only the current graph is published to the client
* creates “notify” global function for red in-page notifications
* publishes server DB

tree\_of\_knowledge.css

dbServer.js

* updateNode, updateLink: updates the current db entries by looping over all the fields in the provided objects
* publish / subscribe

tok.js

* nodeData and linkData – array containing all the data for links and nodes [returned or set by force.nodes() ]
* node and link – d3 selections of all nodes and links in the graph
  + the datum of these d3 objects is linked by reference to nodeData and linkData arrays
* use existence of field “source” of linkData to identify it as a link vs. a node
* run button to keep simulation going
* tick: each time-step, update node positions and SVG objects (most forces are implemented by hand here – except for charge repulsion):
  + define a gravity force independent of charge
  + give noise to the nodes to have annealing-like relaxation
  + define soft-max orienting forces for oriented links
  + define the link constraints
  + position all points along the links (3 pts)
  + position each node “group” (tooltip positioning separate) – derivation triangles (need to orient) and other separately
  + nodes are fixed by setting a property nd.fixed=true in each nodeData element. nd.permFixed is the value node returns to after rollover. nd.phantom designates the node is phantom (currently equivalent to being permFixed)
* redraw: create a visualization from the data
  + set link strengths and node charges and other visual attribudes (position, line type, etc)
  + absence of .text field in the entry received from the server flags it as a phantom node
* updateSelection – update CSS classes, both for selected and edited link/node
* Layout math:
  + MetaMath node and link importance:
    - Goal: Importance should be an estimate of the amount of information flow.
    - Node importance is given by the importance of its children (i.e., a node has a lot of information if lots of things come out of it).
    - We “source” information from the leafs to account for the potential children that might not be included in the graph. Nodes with fewest children will be affected the most by such later additions – thus add a value to each node importance that decays with its Nchild. This also serves to decay the effect of such additions as it propagates up the tree.
    - All the information of a node comes from its parents: split node weight among the parent links.
    - We want to split this according to the relative content of the different parents. Options:
      * Let this “content” be proportional to exp[“depth”] of the parent node: its maximum distance to axioms. I.e., the most “interesting” step in the proof is the one using the “latest” result. Node levels start at 1, link level = parent level. Exp ensures that shifting all levels by const makes no difference, keeping with the fractal structure of the graph
      * “content” can also be inversely related to number of children the parent node already has at lower levels. This way most content comes from a “new” node. Fractal structure is automatic in this case.
    - Then we add the weight of all child links to the default node weight.
    - Realizing this requires back-propagating from the leafs.
  + Node.importance = radius in px (scale up template shape); Link.strength = stroke-width in px
  + Node forces:
    - Node.charge = -(importance)^(p+1); Node.chargeDistance = importance \*cnst
    - Charge implemented asymmetrically – view as acceleration
    - (supposedly p=1 here, but empirically p=2 seems more like it..)
  + Link forces: (g=30\*alpha ~O(1))
    - , so adjusting current position automatically changes velocity as well
    - “spring” force:

* + - Orienting force: rotate link by angle

per tick

* + - To get new position, must divide each step by own charge to get the acceleration rather than force (charge = mass here)
  + Scale-invariant dynamics under zoom in/out:
    - If all distances are scaled by factor b (i.e., nd.importance and lk.strength are as well), then to get the same graph layout and dynamics, we must scale all the couplings as:

Charge=mass:

Links:

* + - Orientation:
    - All this applies only if graph has the same statistics at every scale (e.g., average coordinality is scale-invariant).
  + Remaining free parameters: choice of node importance (radius) and link strengths (widths); transDist; link short and long strengths; orienting force; charge (at equilibr. degenerate with link strengths); charge distance; gravity

tok.css

* Tooltip structure (outer and inner boxes)

gui.js

* All functions for gui operations – keydown, mousedown, etc.
* gui.selected – data array for the selected node (linked by reference)
* showEditor
* all mouse interaction functions

popups.html

* Popup templates

popups.js

* Create and manage editing and display popups
* Update DB: update the database for dat.node according to current form field values; redraw tree and redesplay content, selecting the updated node
  + Takes values from popup fields, does not loop over the values in the linkData array

popups.css