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

DB Indexes:

- Nodes
  - graph: "text"
  - zoomLvl:1, importance:1
- Links
  - graph: "text"
  - source:1, target:1
  - target:1, source:1

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

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:
  - 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
    - $a_i = C\frac{q_j}{r^p}$ (supposedly p=1 here, but empirically p=2 seems more like it..)
  - Link forces: (g=30*alpha ~O(1))
    - $\ddot{x} + \frac{b}{m}\dot{x} = \frac{F}{m}$, so adjusting current position automatically changes velocity as well
    - "spring" force (3 options):
      $r_{i+1} - r_i = -\frac{g}{50} * lk.Strength^{p+1} \left\{ \begin{array}{ll} (x - minDist) & x < transDist * lk.Strength \\ lkStr * lk.Strength & x > transDist * lk.Strength \end{array} \right\}$
      - the "long-link" regime is not needed once we set link strengths diversely enough – can reduce to linear springs
    - Alternatively, can transition between the "weak" (const force) and "strong" (spring-like force) in a different way (other than based on spring length). Want to create a "percolation" regime: where a tree of strong links spans the graph, and gives its primary structure. This can be done by, e.g., choosing the shortest child and parent link for each node to be strong:
      - Keep shortest spring length stored in each node
      - Going through links compare against it: if it's shorter, replace, if it's one that was shortest before, update node length to its (to allow increasing length)
      - Alternatively, if it's shorter, replace, and grow the number with each tick.
      - Also add a margin that all links xx times longer than the shortest link are also considered short. Set xx with Graph_density parameter
      - Think of this as a "strain" on each node, and strong links are cracks that release it – to make the percolation analogy better

        - Orienting force: apply a rotation force propto angle $R = -g * \frac{lk.Strength}{|\vec{x}|}\ (lk.strength)^{p+1}(e^{-\frac{x_2}{|\vec{x}|}} - e^{-1}) * sign(x_1)$ per tick (this way rotation force is length-independent
        - Second orienting force: for unoriented "theorem" links, we want nodes to be roughly at the same level, not far ahead of each other (considered equivalent) – so set force: $R = g * \frac{lk.Strength}{|\vec{x}|}\ (lk.strength)^{p+1}\left(\frac{x_2}{|\vec{x}|}\right)^2 * sign(x_1) * sign(x_2)$
        - 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:
        - $[\mathrm{x}] = -1;\ [\mathrm{t}] = 0;\ \ddot{x} = a = \frac{F}{m} \Rightarrow\ \ [a] = -1, [F] = -1 + [m]$

        Charge=mass: $-1 = \left[\frac{C\, m_2}{r^p}\right] = [m] + p\ \Rightarrow\ \ [m] = -(p+1) = -2\ for\ now$

        Links: $-1 + [m] = [k\ r] = [k] - 1\ \Rightarrow\ \ [k] = [m]\quad and\quad -1 + [m] = [f]$
        or: $-1 + [m] = [\kappa\, r^2] = \kappa - 2 \Rightarrow [\kappa] = [m] + 1$
        - Orientation: $F_R = R\begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \Rightarrow -1 + [m] = [x] + [R] \Rightarrow [R] = [m]$
        - Local gravity: $F_G = G\ (\vec{x} - \vec{x}_0)^g \Rightarrow [G] = [m] - 1 + g \Rightarrow \left[\frac{G}{m}\right] = -1 + g$ (g=2 now)
        - All this applies only if graph has the same statistics at every scale (e.g., average coordinality is scale-invariant).
        - Implemented via forward time-integration: $x_{i+1} = (x_i + v_i\ dt) + a\ dt^2$
    - 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

loadMetamath.js

- parseMetamath – parses MetaMath file (e.g., set.mm) into an array of theorems (each is an object)
    - read file character-by-character triggering flags upon seeing control characters, and running word and phrase accumulators
- loadMetaMath – loads this array into the server DB on Meteor
    - MetaMath node and link importance – from graph structure (done in dbServer.js)
        - 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 (how often it's already been used). 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.
    - This works well, but has a few problems: early nodes get more weight than late ones – so late key results don't show up at 0th zoom level; single parent of an important child is automatically important, even if it's just some small intermediate lemma – but at 0th zoom, we really want one node per "field" – determined really just by nodes with many children. Possible solutions:

        - Divide each node by const = (ave # of children)/(ave # parents) – this ensures that late nodes have on average same weight as early ones
        - Divide each parent link by Log(# other parents +2) – cuts the info going to the parent by the info contained in the proof
        - Divide info from each child link by its child node weight before summing to find ndWt
        - “only display weight for any information once” – the total of child links is split between size of node and weight of parents (so information either shows on the node, or goes into parents – not both).

dbServer.js

- updateNode, updateLink: updates the current db entries by looping over all the fields in the provided objects
- calcEffConn – compute and store effective links at different zoom levels
    - At any given zoom level, show only the 20-50 (VisNodes variable) most important nodes in the given window
    - begin with the goal of capturing the correct ancestral relations – this can be accomplishing by again envoking the “importance (or information) flow” idea. I.e., between two indirectly connected nodes, the effective link should have the strength given by the total “flow” that goes between them in the full network.
    - If w_{ij} is the link strengths, giving also the importance flow, then the total flow between two nodes is:
    $$w_{ij}^{eff} = \sum_{paths\,\{i\to j\}} w_{ik_1} \frac{w_{k_1k_2}}{a_{k_1}} \frac{w_{k_2k_3}}{a_{k_2}} \dots \frac{w_{k_nj}}{a_{k_n}}$$
    where a_i are the node weights, given by the total importance that flows in (or out)
    - This is we consider the matrix Z whose columns are just repeated vector a_i, and let W be the connectivity matrix w_{ij}, then
    $$W^{eff} = Z.* \sum_{n=1} \left(\widetilde{W}\right)^n = Z.* \left(\widetilde{W}\,\left(1-\widetilde{W}\right)^{-1}\right)$$
    where .* and ./ are element-wise (per Matlab syntax) and $\widetilde{W} = W./Z$
    - Now, since we need only to know the effective connectivities between the more important nodes, we don’t need all of W^{eff} but only its part.
    - So, build a sparse connectivity matrix, sorted by node importance (high to low)
    - Then split it at splitN into a 2x2 block matrix $\widetilde{W} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$, such that A is the part corresponding to the remaining important nodes. Since links among A-nodes will be shown explicitly, we need not consider them in the effective calculation, and simply add them in at the end. Thus using block-matrix inverse formula (https://en.wikipedia.org/wiki/Block_matrix#Block_matrix_inversion) we get
    $$\left(W^{eff}./Z\right)_{A,A} = A + B\,(1-D-C\,B)^{-1}C$$
    - We can thus, for a given graph, choose n zoom levels, and iterate this procedure n times to construct ever-smaller effective graphs, starting from the full microscopic connectivity

    - ISSUE: for this to be a good method, reducing the connectivity matrix this way twice in a row should be the same as reducing it once by the full amount. I tried this in Mathematica (as the expressions are very long), and it does not simplify to prove the equivalence – but expressions remain very long, so it's not totally clear. Still, we use this for now as it is better then nothing.
    - Iterate exponentially splitN = lastN / zoomStep^2 such that if the graph is indeed fractal structure, then any visible window will have roughly the same number of nodes at each linear zoom step
    - After the effective links are found, include their values as fields in the Links database – this way each link has fields strength (for microscopic), strength1 (zoomed out 1 level), strength2 (2 levels), …. , strengthn (fully zoomed out, leaving only 20-50 nodes)
    - The relevatn fields are then loaded into strength value of linkData array on the client in tok.js
- maxZoomLvl – find the most zoomed-out level in a graph (since it will depend on graph size..)
- publish / subscribe – implements zoom behavior, publishing only the few relevant nodes at a time according to the parameters passed into subscribe from the client
    - this is called on each graph redraw() in tok.js
    - subscribe uses the currently visible window coordinates visWindow, and current zoom level currZmLvl stored in Session variable
    - it first calls to publish the visible nodes:
        - find all the nodes at currZmLvl inside the visible window. If this is is between 20 and 50, then we're done
        - If it's too many, then zoom out: increment currZmLvl (thus taking only the large nodes and effective connectivity at that level), then check number of nodes again
        - If it's too few, then zoom in: however, since the smaller nodes might not have been positioned yet, we can't use their positioning inside visWindow to find which ones to show. Instead, we use the graph structure. We find all the children of already visible nodes for whom the visible node is its strongest linked parent. Similarly, we find all parents of visNodes …. strongest linked child. We then loop this algorithm until no new nodes are added – adding only nodes at currZmLvl (else it won't terminate). This ensures that all the (linked) nodes can be reached. New unpositioned nodes start off next to their previously visible parents/children.
        - We then loop the zoom in/out procedure until we get number of nodes between 20 and 50, unless we overshoot, in which case we also stop.
    - We then publish the phantom nodes that keep the visible graph structure close to what it would be in the full graph. Phantom nodes are fixed and mostly outside the visWindow.
        - Need to get the right spring-forces and right charge forces
        - For the springs, we should really include all visible-to-other links, but that can create too many links for efficient running. Can include 20 most important links – but then we can end up with disconnected nodes. Instead, choose at most 2 most important phantom links per node. After this, if a phantom nodes happens

to not be positioned yet, then don't publish it (the resulting orphaned links are filtered out in redraw() on the client).

-

MetaMath network analysis (graph_analyze.m)

- The degree distribution of children is scale-free, with $prob(c) \propto c^{-0.9\pm0.1}$
- Degree distribution for number of parents is instead log-normal: $prob(\ln[p]) \propto \exp\left[-\frac{(ln\,p-3.3\pm0.2)^2}{1.5\pm0.05}\right]$ (except that for small p < exp(3.3), discretization makes it look different – though the integrated weight seems correct)
  This distribution remains constant after ~70 000 nodes, up until which point the mean rises (linearly, or logarithmically – doesn't really matter)
- Node weight distribution from standard algorithm is also scale-free with same 0.9 exponent