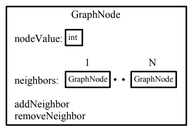
Connect "MyMath" namespace



Implement in C as a data struct.. not sure how want to attach the functions.. all structs, I suppose, connect to the same manipulators, which get passed a pointer to the data struct..

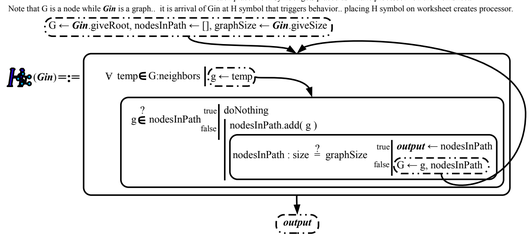
Let's see.. Idea for from-into is to have a thing that accepts in-boxes of the From, and does the extract and send to each of the patterns inside the Into.. so a single entity exists for the from-into operation.. that one entity is both the from and the into.

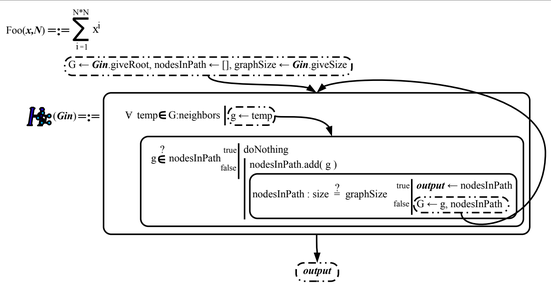
What that means for the Hamiltonian path thing..? For that, let's see.. have the rounded boxes.. each is a processor gets created, with innards being the behavior spec.. the arrival of a namespace triggers creation of a new processor, connected to the namespace..

Let's create a box-type code that contains From-Into symbols, and has patterns-within-patterns, plus simple Fn calls on data-structs, and patterns passed in namespace that get hooked in somehow.. maybe consume symbols defined on worksheet.. by which some get connected pattern-in-inbox-of-pattern style.. and srcHolder backed processor created elsewhere, persistent, is hooked into worksheet (import and attach to pattern symbol?)

Note, placing things onto worksheet creates elements.. those elements handle triggering activity, creating processors that wait to be triggered, being triggered by changes in worksheet namespace assignments, and so on.. For example, Placing def of Hamilt symbol creates a new entry in worksheet namespace, which has the H symbol and the src holder of the syntax graph assigned to it (or something like that.. the definition creates a from-into pair.. can handle that pair via src manipulation or by creating a FromInto struct in each place the H symbol gets placed as "make processor" on the worksheet.

Note that G is a node while ***Gin*** is a graph.. it is arrival of Gin at H symbol that triggers behavior.. placing H symbol on worksheet creates processor.





H(*t*) := Taylor of *z* + **Jacobian**(z) with respect to *z* about 1.213 for 4 terms ⇒ <*pattern* (*t*)>

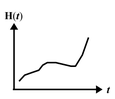
t := 8.915

make H(*t*) into a processor ⇒ **H**(*t*)

H(*t*) ≟ 2.7189

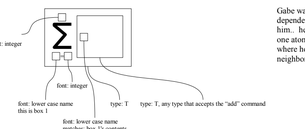
t := {1,2 .. 20}

H(*t*) ≟ {2.14, 3.919, .. 21.78}

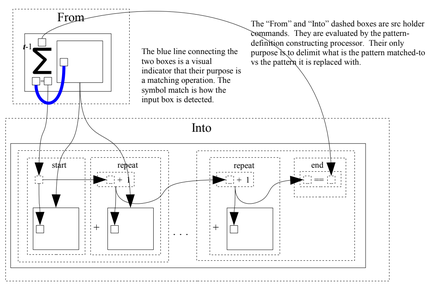


Summation

Summation is defined as a series. Series are translated into Loops.



Gabe wants to see what he's saying, and the dependencies between the index terms bothers him.. he wants to be able to say "this happens in one atomic action".. he wants to have a primitive, where he states the relationship between the neighboring

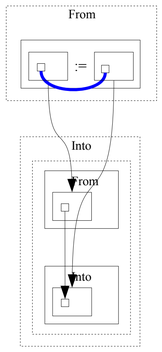


The blue line connecting the two boxes is a visual indicator that their purpose is a matching operation. The symbol match is how the input box is detected.

The "From" and "Into" dashed boxes are src holder commands. They are evaluated by the pattern-definition constructing processor. Their only purpose is to delimit what is the pattern matched-to vs the pattern it is replaced with.

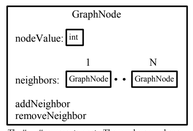
Specification of Define

The line connecting the two boxes is a visual command to search for all groups of tokens that are of "parameter" tokenType and have the same characters.



Pairs found by the blue-line command are connected by arrows

In Hamiltonian path, get a graph.. so
have graph data structure
Have a root of the graph.
have neighbor of each node
Q: how to define a data structure? What does it look like?



The " . . " represents a set. The numbers up above indicates that it is an ordered set. The "GraphNode" inside means that each element is of type GraphNode. The " . . " is a pattern that was inserted via the GUI

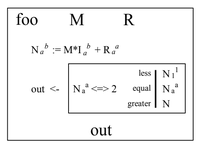
Notice that defining a data structure is syntactic sugar for defining a processor. Creating an instance of a data structure is the same as creating a processor. One passes the source for the data-structure to the creator. A data structure definition has implied commands, one to access each element.
Other thing is that one doesn't have to define the type of the contents of fields.. the type will be inferred by use.. if it can't be inferred, then the programSpace causes an interaction with the Display and the programmer is asked to fill in.

In fact, a data structure in EQNLang is essentially the same thing as an object in OO languages.
The functions that the memory processor can perform are stated inside the data structure, but defined outside on the same worksheet.

" := "

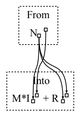
" := " is syntactic sugar (a preprocessor command). What's on the LHS becomes the contents of a "from" box. What's on the RHS becomes the contents of an "Into" box. Further, each italic name on the LHS is matched against each italic name on the RHS. For each match, the names are removed and turned into boxes, then an arrow is placed from the box on the LHS to the box on the RHS.
Any arbitrary expression can be on the LHS, and any expr can be on the RHS.

when an expr in source matches multiple LHSes, alert the programmer during edit, throw error during consolidation (or something like that)



$N_a^b := M*I_a^b + R_a^a$

The expr above turns into the custom notation def below:



Properties Attached to Data

Learning about R language, which has properties attached to data structs, and syntax that operate on the data differently, depending upon the contents of the attached properties.. so, a list element can contain an object of any type, and an operation could be to select all list elements of a particular type. Likewise, matrices have column and row properties attached, and operations can fetch the contents of those properties explicitly, and use that in the operation performed on the matrix contents.. or even modify the column and row properties values.

So.. seeing this as the equivalent of the property chains attach to nodes and links in a syntax graph. The point is that it's data ABOUT the data.. each property is attached to one portion of the pattern that data is arranged within.

Couple things.. one is that the properties can be used to pick the way in which the data is combined with pattern to generate an active pattern (see Dissertation). For example, the attached property could say the 32 bits of data is to be combined with integer-add pattern, or combined with floating-point-add pattern, or with MIPS instr-pattern.

Second is that can have hierarchy – leaf-level pattern has its particular pieces each attached to a higher-level pattern, which in turn has its pieces each attached to yet higher-level.. and so on.. the leaf-level has this same concept, except that its properties and patterns are fixed. seeing this as embedding a data-structure into a node in a syntax-graph. The nodes and links of the graph define a data structure, and the properties attached define how higher-level patterns activate the data-structure, turning it from data into live, the way putting 32 bits into an integer add turns both the bits and the add-circuit into something live.

So, the leaf level equals a fixed pattern with fixed values for properties. Hence, the property values don't need to be read in order to figure out what pattern to combine the data with to make it live. Instead, the decisions about patterns to combine with is fixed. That's what a data struct in C does – it fixes property values, such as number of fields, the type of each field (int, float, pointer), and so on. In R, these properties are read during the run, and behavior depends upon the value read.. in C, these properties are fixed, and behavior on the data is fixed (behavior == which pattern the data is combined with to make a live thing).

So, in POP, want both.. and want ability to turn variable into fixed at any point desire.. so, seeing something like passing around syntax-graphlets, with syntax-graphlets as the content of nodes.. at the leaf, instead of a lower-level syntax-graphlet, have a block of data that has a single ID in prolog, which identifies what code-patterns can accept that data-block. Some code-patterns have a fixed pattern that the data-block is seen as, so the contents of the data-block don't have, in turn, their own Ids, rather the ID is fixed into the code-pattern.

That means that a syntax-graphlet can be "compiled" – which causes a new ID to come into existence – that ID is attached to the compiled pattern. The syntax-graphlet, therefore, has to attach patterns that take the chunk of data in a particular node or particular property attached to node or link, and extracts particular sets of bits and puts those into particular behavior-making patterns (like integer-add).

Seeing maybe use a 64bit int as the ID, and use the least-significant bit to indicate whether syntax-graphlet or ID.. so, zero mean syntex-graphlet, and the bits are interpreted as a pointer to the root of the graphlet. 1 indicates pre-compiled graphlet, and the bits are used as a pointer to the compiled version (the 1 is masked to zero first). Something like that.. will figure out details as create the first hand-compiled version of POP..

Here's what I'm thinking.. fields of a data structure can only be accessed via accessor-functions.. in the code, that would be the "dot" syntax, such as in the Hamiltonian, where it does nodesInGraph.add(g). The other thing, though, is that other patterns can specify that they accept data arranged according to a particular syntax-graphlet or related syntax-graphlet.. sooooo.. can those then only use accessors? Or, can they directly operate on bits within the chunk?

Right.. now getting down to the type-system stuff.. inheritance, and sub-types, and all that..

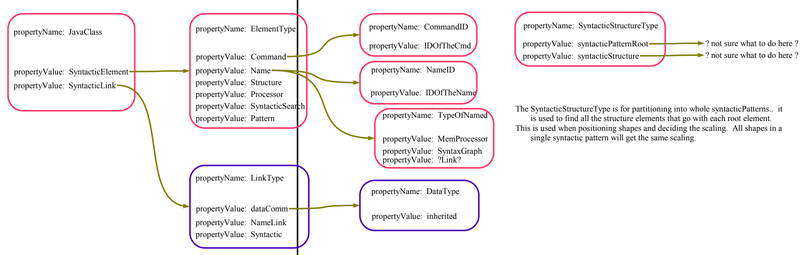
Q: what going to do for hand-compiled version?

(Was seeing properties attached to processor being used to find processors, and seeing some form of hierarchy in system.. processors that compute on things **about** other processors.. so, compute on the properties, namespace connections, and so on.. is there any value to this? Is there value, say, to processors that can manipulate the namespaces of other processors, detaching and reattaching elsewhere.. ultimately want an analog of human consiouness record and playback.. and ability to make patterns **about** other patterns.. This thing of noticing common properties, and forming a pattern that is the collection of patterns that share those properties.. then correlating observations with members of that collection, to gain association between collection and behaviors, or other associations.. that's the nut to crack.. ability for one processor to notice things about another processor and generate the specification of a new processor and create it and hook it to the under-observation processor, so that it becomes a part of the system..)

Notice that variables are identified by an ID. The Visualizer keeps the visual representation of each variable. The Modifier receives data about the visual appearance of each variable from the Display (eg a string of the variable name). The data comes inside a GUIGesture sent from Display to Modifier. The Modifier must relay this information directly to the Visualizer. The Modifier does not keep this information, nor does the syntax graph hold any indication of the visual appearance of variables. Many variables are visually represented by strings, so the Visualizer would store such a string. The font of the variable, however, indicates properties of the variable. For example, a memProcessor has a different font than a level-below SyntacticVariable. Each kind of variable is rendered in a different font, with different effects. The font and effects are chosen by the Visualizer, and correlate with the properties of the variable. The properties are stored in the syntaxGraph.
Likewise, the visual representation of a command is also kept inside the Visualizer.
Notice as well that when implemented in Java, for example, all property names and all property values can be enumerated types. One complication is that some command names are custom and generated during entry of a syntax graph, and the variable IDs are all generated during entry of the syntax graph. Thus, in Java, propertyValue is likely to be an integer that is correlated to an enumerated value. The integer will be used in a switch statement. There will be a fixed number of propertyNames for EQNLang, so that will be an enumerated type.

Hierarchy (Dependencies) of PropertyNames and PropertyValues in POP

t.



The SyntacticStructureType is for partitioning into whole syntacticPatterns.. it is used to find all the structure elements that go with each root element.
This is used when positioning shapes and deciding the scaling. All shapes in a single syntactic pattern will get the same scaling.

Treat constants as mem processors. So, for example in custom Tensor notation, the index-column is a Structure ElementType, and so is the Tensor Index that goes into it. That Tensor Index has a link to whatever is seen at that point. It could be a name that stands for a processor that will be sent from somewhere else.. in this case it is a Name ElementType, with a dataComm link to whatever syntacticPattern sends the data. Sending data is sending a processor containing the data.
Or, the Tensor Index could be a name that attaches to some other bit of syntax. This is the thing where have a complex expression and just assign a syntactic-level name as a short-hand for that expression. (What about syntax-embedded-in-writes? This is what SyntacticSugar is in Scheme, and what a macro is..)
Or, the Tensor Index could be a constant. In this case the element is a Processor. Will have probably a property for Processors, that can put in the syntax directly, that they are a constant-containing processor, and what the constant is.
Pattern I'm seeing as a from-into pattern.. it's not a command, because command is the name attached to that pattern.. so need a way to say "this is the pattern itself, the thing attached to the command-name" (where attached means correlated-to)

thinking "Syntactic" Link is for syntactic-name and for syntax-level rewrites (need some way of getting at the concept of srcManipulators that are embedded.. ie, some code specifies a re-write to other code that takes place before the other code reaches the translator. Some sort of notion of "when" the re-writes happen.. the re-write is turned off for the code that specifies the re-write to be performed on other code.. Same notion as in the different kinds of variables.. one kind is used to match, to specify the from-into pattern.. the second kind of variable, sitting right next to it, is not used when detecting matches. There could be two of those also, but they are not looked at by the match-detector. However, after the match detector is done, have a from-into pattern that has those second kind of variable in it.. NOW those become active, when the re-writer kicks in. The re-writer will see those and perform substitutions. So, the distinction is which processor treats them as semantic info vs flat data. This is what "*-time" means, the * is the name of the processor that performs the manipulations. "Edit-time" is the editor, "run-time" is the animator, "install-time" is the install-processor. So seeing some syntactic-notions meaning "before normal re-writer", this is the macro-notion.. something that operates on the source code as data devoid of semantic meaning. Whereas the re-writer treats the source as does have semantic meaning.. that's the distinction between macro-rewrite and "normal" re-write

More on What is Syntax

Summarizing:
Visual pattern correlates to action pattern. The syntax pattern is a middle step between the two. The syntax pattern has a feature for each visual feature that affects the action pattern (when action pattern is animated).
The syntax states exactly the visual pattern. The same visual pattern can correlate to very different action patterns for different languages. For example, in Pascal "A := B + 1" correlates to an action that is observably different than the same visual pattern in EQNLang (consider the variable properties.. int's in Pascal, but could be Tensors in EQNLang)
The semantics states exactly the action pattern that the visual pattern is correlated to. Each language correlates a given visual pattern to different action patterns. Semantics is to action-pattern as syntax is to visual-pattern.
So, the purpose of syntax is visual. The requirement of syntax is have a feature for every visual feature that can correlate to an action pattern feature.
And that, in a nutshell, is syntax.
So, can have a single universal set of syntax patterns that is capable of capturing every feature of correlation in a visual pattern.
That, then is the task to accomplish.
God this is great, after all that fear, and a couple weeks ago coming back to this, feeling completely inadequate and incapable, it feels great to finally have this nailed. To have a set of underlying patterns that are consistent with absolutely everything. Sweet. A small, simple set of patterns that fit with All. I love it. (something about that.. finding a small set of simple patterns that fit everything, that are CONSISTENT with everything.. something about that triggers this profound pleasure center for me.. recognizing when I have such a set of patterns underlying details I've been wading through.. that recognition triggers a pleasure pump in my brain.. THAT is a part of why I do this work.. and it's interesting that it's there.. have a feeling that many people don't have that trigger.. and those people won't choose to pursue finding such sets of patterns either.. Interesting from the standpoint of Inorganic Life.. what "should" they have in them.. what emotional mechanisms.. where "should" == I want the II. to match as high on some goal-pattern that I have as can arrange things.. so "should" means that the "should"ed thing (action here) will move II. higher in the goal-measurement.. but not really clear on that goal pattern yet.. it scares me a bit looking there.. is it a power thing? I think so.. in much part.. a "father" thing also.. creating life (which has a power component).. yeah, and a name.. that seems to be lurking within me.. being the Archimedes of these times to the future.. I feel like that's a built-in emotional mechanism thing.. (also interesting in itself to II.))
Back to visual elements.. the visual elements-of-correlation:
-- shapes
-- pieces belonging to a single pattern
-- multiple levels of patterns (full pattern being element of another pattern.. strictly visually)
-- can have a shape be a piece of one pattern
-- can have a pattern that has a given shape as part of it in turn be itself a piece of another pattern
-- can have a single shape in single spot be a piece of multiple patterns (namespace pattern as well as command pattern)
-- visual position of shape indicates:
-- -- inclusion as piece of pattern (ie, this shape is in this pattern over here instead of that pattern over there)
-- -- position within hierarchy of patterns, via inclusion directly within one particular pattern that is itself in the hierarchy
-- -- visual position can indicate a single shape's direct inclusion in more than one pattern at the same time
For example, consider a page of math. At the top a variable is defined. Down below a variable of the same name is used. The use of the variable in that position places the variable both in a namespace pattern, which is how the semantic meaning of the variable is determined, and at the same time places the variable in the usage pattern. The syntax should capture the inclusion of the use in both of those patterns. One way is to have the syntax map the positions of each shape relative to the other shapes. The other way is to use implications from the semantics of the language to determine any links such as the namespace link. Looks like going to do this option.. use the GUIToCommandTranslator to figure out all the different syntactic patterns to associate a given syntacticElement to.
So, in the syntax data structure, make a link be the embodiment of visual inclusion. Make a link to each pattern-root that includes a particular syntactic element as a direct member of the pattern. A link is the equivalent of visual inclusion.
Let's see.. want an inclusion link for entire sub-pattern, so that's a link from the syntacticPatternRoot. Want an inclusion link from each and every shape-containing syntactic element?
Seeing kinds of things that work across languages..
-- have node that has a shape directly in it..
-- have a node that is only a carrier and organizing point for other nodes (TensorIndexColumn for example)..
-- have a node that is the root of a single pattern..
-- have a node that represents interaction between patterns? IE, transfer of animator in a sequential language, or transfer of data in a data-flow language (like math notation).. must include this.. some languages have visual elements that directly correspond to interaction (like BaCTiL for example, and other large grain data flow languages). Some languages use physical location on page to represent this interaction, other languages have explicit visual shapes that indicate this interaction. So, make it an interaction syntactic element, without implying the nature of that interaction.. for example, consider Linda or Vitria, which would represent channels with syntax, and have syntax that indicates interaction with such a channel.. in this case the interaction is posting a match pattern and accepting a match back, or sending a message.. so could have a visual representation of send a message, which is an interaction, so make an interaction an explicit syntactic element type. Or, want to play it safe and just make interaction be indicated as a property.. so use a standard syntactic element as the link between patterns, and indicate that it is a link as a property with the propName == "link" and the propValue == <the link value>..?. Which way want to start? Start with the more-work method (everything is a syntactic element), or have a little bit of built-in specialization (multiple Java types of syntactic element)
Okay, quicker if I just start with a single Java type: syntacticElement.. then make links be a property-value, and make interaction be a property-value, and so forth..
Here's what I'm seeing: interaction-links have "dataSentTo" or "dataReceivedFrom" property name, with pointer to receiving-port or syntacticPatternRoot syntaxElements, where receivingPort and sytacticPatternRoot are themselves properties attached to the target syntactic elements.
The properties will be handled by big switch statements.. can even make the propertyNames enums, one enum set for each language.
So, here we go..

Syntax Graph for a Simple Case

Have two kinds of structural syntax nodes: elements and links, plus property nodes that are attached to an element or a link
Each kind of structural node has a list of the node's properties attached.
An element has two lists of links, one list for incoming, one list for out-going,
An element also has a list of the properties of the element
An element may optionally also have a list of sub-elements, which together make up a single syntax pattern, as a single unit – for example, one of Gabe's rotation pictures will have several elements, each a visual piece such as a swirl or the pointed line being swirled.. it seems that the meaning will group several pieces together as a single operator, together with parameters of the operator.. something like that..

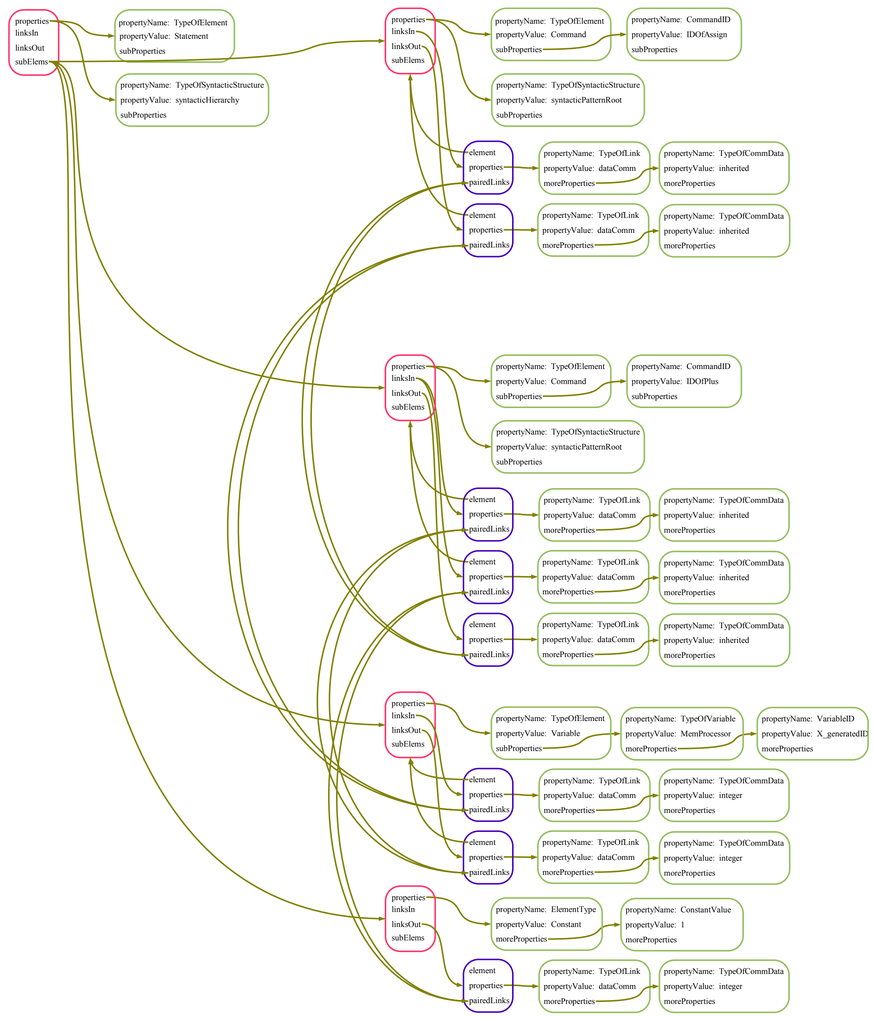
A link has a type, either an in-coming link, or an out-going link.
A link has a pointer to the element it is associated with
A link has a pointer to more links in the list
A link has a list of the link's properties
A link has a list (array) of other links that it is paired to (an incoming link of one element is paired to an outgoing link of a different element – can be one-to-one or many-to-one or one-to-many or many-to-many)

A property node has a property name
A property node has a value from the named property's set
A property node has a pointer to the next property
A property node can be context sensitive, and may refer to previous nodes in the property list, even by offset of where they are relative to self

Red round-cornered boxes are element nodes, blue round-cornered boxes are link nodes, green rounded boxes are property nodes, and arrows are pointers among nodes
In practice, it turns out to be more convenient to inter-mix type information with the syntax information. The structure of the graph remains essentially entirely syntactic, but some of the properties added have a type information role. For example, type information may be used to choose the visual shape displayed for a link. Type properties are included in the syntaxGraph, in part, because they are used in the grammar. The grammar is implemented cooperatively by the Visualizer, Modifier and Display, which all work together to only allow building a syntax graph that is consistent with all grammar constraints.
In additions, the syntax information is a bit removed from a strict visual correlation.. the Visualizer acts as a translator between the syntaxGraph and the pure syntax (which is one-to-one with visual). The syntaxGraph is thus a bit closer to semantics.
For example, the syntax-graph states that an element is a command, and states the name of the command. But it is the Visualizer that determines which shape will be used to draw that command. So shape information does not appear in the syntax graph, only in the DisplayList.

Now, a simple example of a snippet of syntax. This snippet is currently disconnected from anything else, so it has incoming links that are not paired to the outgoing links from something else. This syntax graph should be able to support any language's syntax, hopefully.. may have to modify this base structure if discover cases that it doesn't cover.
For now, the basic POP syntax graph is centered around "syntactic root" elements. Each of these has links coming in and links going out. The incoming links are the inputs to the processor, and the outgoing carry what the processor creates. For example, in Java, a method would be a syntactic root, and the inputs to the method would be incoming links, while the return value would be the outgoing link. A method body is a root element that has a number of sub-elements, one for each statement in the body. The sub-elements are ordered. If a method calls another method, then a "call" root element is placed as a sub-element of the calling method's body. This call root-element has an outgoing link for each argument sent as part of the call, and it has an incoming link for the return value. These links are paired to the corresponding links of the called method. Question: duplicate the called method's nodes? Or, simply allow many different callers to pair up to the same called-method's links? Maybe add a field in the links for "paired links". That way, the in-coming links of a method can have a whole list of corresponding out-going links coming from call-elements that are in the bodies of other methods. When a call is added to the code, no links are duplicated, and the called method isn't duplicated, but rather new pointers are added to the paired-links lists.
A variable is never a root element. It is always a sub-element of either the global context element or, in Java, a sub-element of a class element (but a "class" is actually a context?), or a sub-element of a method body (where a method has its own context as part of what a method is). A variable element has an outgoing link, which represents reads, and an incoming link, which represents writes. These pair to the corresponding links of any statements that read or write it.
Notice that in many cases, variables just end up being names of wires, and the reads and writes end up reducing to pass-throughs. It seems to me that the only case in which a variable acts as anything besides a wire is when timing is considered, and the future cannot be predicted. When it is unknown whether a future processor may want to read a given value, then the variable cannot be reduced to a wire or pass-through. Ahhh, right, this is data-flow – variables are all wires.. but it has a hard time when it is unknown whether a given value might be wanted by a future portion of the dataflow graph.. that value has to be recirculated around and around, waiting to be replaced by a different value to be recirculated or else to be duplicated and sent off to a graph element that wants the value. Right.. so, sheds some light on variables – they are recirculated values! In fact, that is how they are physically implemented as well! Static ram and latches are, physically, recirculated values, while DRAM and CD-ROM and hard-disks simply have a long recirculation time before the configuration decays and has to be renewed (the value is copied off by an external thing, such as a read-write head). Good. Persistent state is recirculated (renewed) – seems like a fundamental thing (would apply to physics – cycle equals "stateful" particle).

X = X + 1;



∫