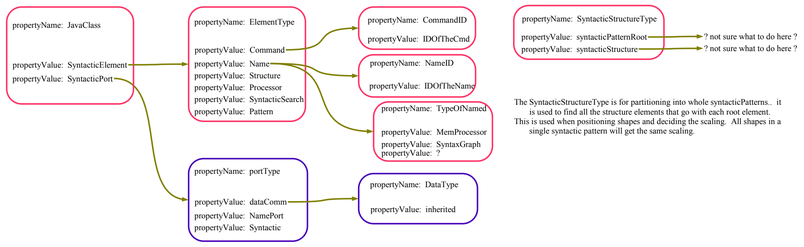
Hierarchy (Dependencies) of PropertyNames and PropertyValues in POP

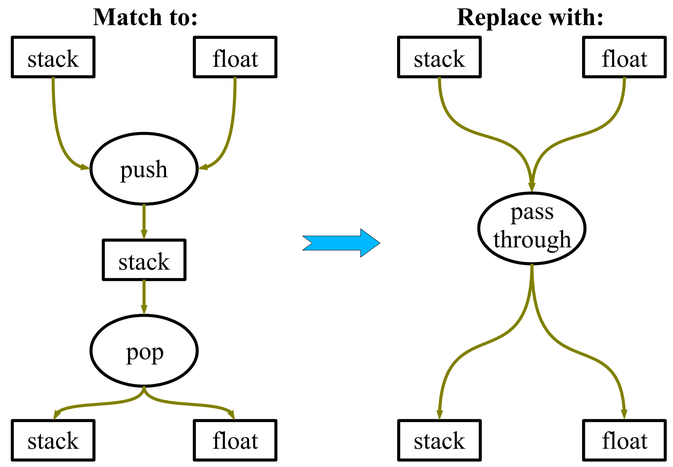
t.



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.

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 port to whatever is seen at that position. The port could pair to a name element, where the name stands for a processor that will be sent from somewhere else.. in this case it is a Name ElementType, with a dataComm port paired to whatever syntacticPattern sends the data. Sending data may be sending the name of 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 re-writes? This is what SyntacticSugar is in Scheme, and what a macro is..)
Or, the Tensor Index could be a constant. In POP, a constant is a Processor, just like a variable (memProcessor).
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)

Gabe pattern



Syntax Graph for a Simple Case

Have two kinds of structural syntax nodes: elements and ports, plus property nodes that are attached to an element or a port
Each kind of structural node has a list of the node's properties attached.

An element has two lists of ports, 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 port has a pointer to the element it is associated with
A port has a type, either an in-coming port, or an out-going port, which is determined by which list the element connects it to
A port has a list (array) of the port's properties
A port has a list (array) of other ports that it is paired to (an incoming port of one element is paired to an outgoing port 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 list (array) of sub properties
A property node can be context sensitive, and may refer to previous nodes in the property list, even to an offset of where others are relative to self

Red round-cornered boxes are element nodes, blue round-cornered boxes are port 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 has behavioral meaning, such as which ports pair up with which other ports, and some of the properties added have a type information role or even contain values, such as for constants. For example, type information may be used to choose the visual shape displayed for a port. 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 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 is something seen by a programmer. A root may have sub-elements that go with it. For example, in Tensor Notation, a tensor has columns next to it, where either numbers or letters or other syntactic entities go. There is a top position and a bottom position in each column. Hence, the tensor is the root element, with each column being a sub-element, and each column has two ports, one for the top and one for the bottom.
For a syntactic root that represents a processor, such as when syntax indicates a command to be performed, the incoming ports 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 ports, while the return value would be the outgoing port. 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 within the calling method's body. This call root-element has an outgoing port for each argument sent as part of the call, and it has an incoming port for the return value. These ports are paired to the corresponding ports of the called method. This allows many different callers to pair up to the same called-method's ports. Incoming ports of a method can have a whole list of corresponding out-going ports coming from call-elements that are in the bodies of other methods. When a call is added to the code, no ports are duplicated, and the called method isn't duplicated, but rather new pointers are added to the paired-ports lists.
A variable is always a sub-element of a context element, 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 port, which represents reads, and an incoming port, which represents writes. These pair to the corresponding ports 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;