# An Extensible Term-rewriting Based Language

by Sean Halle

*Email:* `seanhalle@yahoo.com`
*Web:* `codetime.sourceforge.net`

*Nov 2, 2007*

**Abstract**

It would be nice to have a language that allows custom notation of any form to be added by the application programmer. This would encourage re-use of the custom notation, improve programmer productivity, increase self-documentation and readability, and enable many activities done by hand in math and science to be more efficiently performed.

It would also be nice if such a language had easy to use parallel constructs, and enabled direct mapping of functions to hardware accelerators and parallel co-processors. The importance of parallelism and the trend of including parallel co-processors on chips encourage such language support.

We introduce such a language. The language has no text-based form, rather the source form is a syntax graph made up of standard data-structures. The language explicitly represents active-entities, called "processors". It has three separate mechanisms to extend the language with custom notation. The first is a fairly straight forward adaptation of term-rewriting. The second uses active entities (processors) in the development environment to generate the term-rewriting rule. The third introduces an explicit symbol for the creation of a processor from source code, and defines a standard processor type called a "sourceManipulator". SourceManipulators take in a syntax graph and produce a syntax graph as output, which can then be sent through a creator symbol to create a processor.

## 1 Introduction

Programmer productivity, reuse, self-documentation, ease of maintenance, and parallelism support are important topics in languages at the moment.

## 2 Background: The Environment Surrounding EQNLang

Languages assume some form of environment that provides services such as long-term storage, communication primitives, and the means to "run" a program. The environment that EQNLang assumes is a fairly new one that has been designed for portability across parallel hardware configurations called CTOS, short for "CodeTime OS".

The main abstraction in this environment is "processor." Everything is considered a processor [cite CTOS paper] including "files", running pieces of code, and the things that provide OS services. For example, the equivalent of a file is a processor that contains structured data. Accessing the data requires connecting to the file processor and engaging in a conversation.

One important distinction in this environment is between code and processor. Code is data that lives inside a "file" processor. This data is used to create a processor. The created processor exhibits the behavior the code specified.

With this view in mind, when one considers an OS, it is clear that an OS is a body of code, while a running OS is a processor created from that code. Thus, a PC has a disk filled with data that is the Windows OS. When the PC starts, boot code creates a processor from that data. Thus, a new instance of an OS processor is created each time a PC boots.

Moving on, a processor can contain other processors. In CTOS, most of the OS services are accessed via a set of "built-in" service processors that reside inside the CTOS instance. These OS-service processors are: the creator processor, translator (compiler) processor, name-discovery processor, and so forth. To obtain an OS service, a connection is made to the appropriate OS service-providing processor, then a converstation is engaged in with it.

The OS's creator processor is used to create new processors from code. To create a processor, a connection is made to the Creator and code is given to it. From the creator, out pops a new processor that exhibits the code-specified behavior.

In CTOS, a program is more than just the collection of code. A program is a live system of both processors and code. The processors provide many of the behaviors one normally associates with a language such as namespaces, scoping, and compiling. The code data is held inside "srcHolder" processors, which can be thought of as file processors specialized with extra code-specific functions.

srcHolders are in turn held inside "programSpace" processors, which have specialized functions needed while developing a program. The programSpace provides some of the functions required during modifcation of code, so code can only be edited when its srcHolder is inside a programSpace. A programSpace serves as a container not only for srcHolders, but also other programSpaces, workSpaces, and automatically created processors.

Many of the IDE behaviors are supplied by the processors of a program. The srcHolders perform all editing behavior. The the visual representation that a human looks at while editing code is generated by processors inside the srcHolders.

The last major category of processor is the "workSpace" processor. These are used to contain other processors. So, for example, the OS processor is a form of workSpace. A programSpace is also a specialized form of workSpace. When a program is run, it normally first creates a workSpace to contain all the other processors the program makes.

In general, workSpaces hold automatically created processors, programSpaces and hierarchically other workSpace processors.

For user interaction, CTOS defines Display processors that reside outside any CTOS instances, instead running natively on hardware with a display. A Display is considered a hardware device. Each kind of Holder processor and Space processor can be asked to connect to a Display. This involves a "Visualizer" processor generating a visual representation of the Holder or Space contents. The visual representation is sent to the Display which paints it, for a person to look at.

A typical visual representation is seen in Figure 1 which shows a CTOS processor instance, a few processors that exist outside the CTOS instance such as web servers on the net, and the inside of the CTOS instance. The shown CTOS instance contains the special OS-service processors, a workSpace, programSpace, srcHolders and generated processors. More explanation of this figure will be given in section ⟨reference|secOS⟩.

srcHolder processors contain a number of specialized processors. These are a modifier, a visualizer, and a syntax graph. In addition, "srcDisplay" processors are connected to srcHolders to display the contents. A srcDisplay is a Display processor specialized to display source code (of any language).

Editing code happens in a loop. The loop is described starting with the modifier receiving modify commands. It performs the corresponding changes to the syntax graph and notifies the visualizer. The visualizer translates the syntax-graph into a list of display elements and sends it to the srcDisplay. The srcDisplay paints the display elements, displaying the source for the user. The user sees this and responds with GUI gestures. The gestures are bundled by the Display and sent to the modifier. The modifier translates the GUIGestures into modify commands, then performs the commands. Then the loop repeats.

## 2.1 EQNLang

EQNLang is primarily a compiled language. However, because EQNLang is implemented inside CTOS, it has fewer features than traditional languages. The missing features belong instead to things other than the language. For example, namespaces are part of the OS rather than the language; scoping rules are part of programSpace processors rather than the language, and so on.

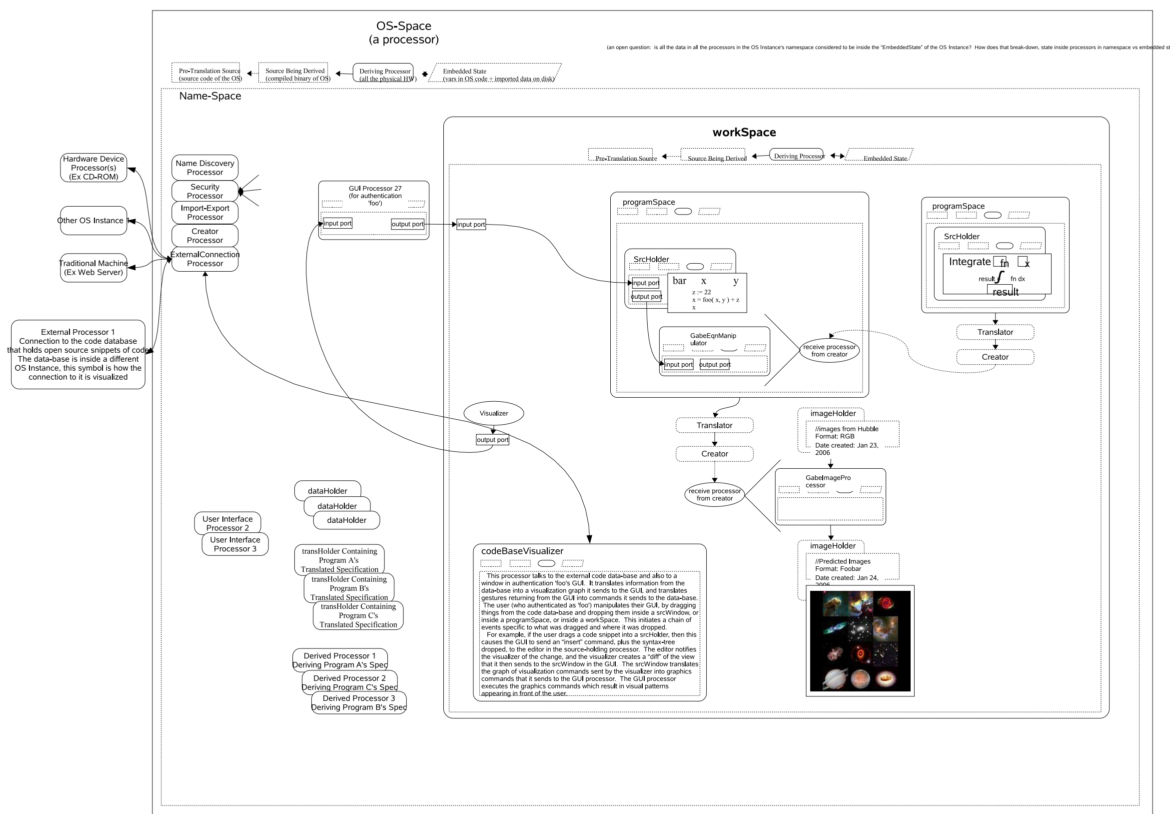


**Figure 1.**

We desire application-code extensions to a compiled language, without modifying the compiler. In general, this is possible in (at least) two ways: by re-writing the extended syntax into language primitives just before compiling; and by doing live modification of source, while the program is "running", then compiling and linking it in to the running program. EQNLang uses both methods.

Because of the nature of CTOS, compiling an EQNLang program is quite different than with other languages. Traditional compiling includes lexical analysis to turn character sequences into a syntax structure, plus grammatic analysis to determine the kind of syntactic element each token represents, plus optimization transforms, followed by translation to a different language. In CTOS, these functions are separated out and performed in many different processors. As a result, only translation is performed in a CTOS compiler. To avoid confusion, in CTOS, what would otherwise be called the compiler is called the Translator.

The re-writing step, in EQNLang, is implemented with standard re-write techniques.

Meanwhile, new syntax that semantically means that source code will be modified "during a run" and turned into a live processor that is linked in during that same run is implemented using features of CTOS. CTOS includes a command to translate code, one to create a new processor from translated code, and one to link the newly created processor into a pre-existing system of processors.

As far as the visual aspects of language extensions, there are many different types of new syntax:

- Simple re-writable custom syntax
- Generated re-write rule custom syntax
- Compound structure custom syntax

- source modifying syntax for the end-user programmer to use (which has three flavors)
- source modifying syntax for specifying how to re-construct a live system
- source modifying syntax used when by-hand constructing a live system "under the hood"

All of these types of new notation must be visualized for the programmer, and they must have a source form for the translator and CTOS.

It can be tricky designing a syntax system capable of doing all this. Four things must be extendable: the syntax-graph structure itself, the visualization that represents the new syntax-graph structures, the grammar that says what is a well-formed statement, and the type system.

Let's look more closely at how these four things that must be extendable relate to each other. Extending a language means adding new "keywords" to it, which is equivalent to adding new syntax. This new syntax must be viewable by the programmer. Because EQNLang's source form is a syntax-graph, the new syntax must come with a way for the programmer to see it mixed in with other syntax. Not only that but the new syntax is used in combination with other syntax; some combinations have a meaning in the language while others don't. So the grammar of the language must be extended in unison with the new syntax to state what ways are valid for combing the new syntax with other syntax. Finally, the new notation may define a new type, or it may be a command that has requirements on what types it accepts, so the type system must be extended to check that communications between commands respect the newly defined command interfaces, while possibly communicating newly defined types.

The difficulty of extending the syntax-graph is handled by recognizing that most features of a syntax data structure are in fact properties. When one thinks about the fields of a data structure, the field name is a property name, and the field contents are values the property can take. Similarly, the name of the data structure itself can be seen as a property-value. In this case, the collection of all possible names of syntax data structures is given a name, which is the property name. Hence, the collection's name is the property name while the name of one data-structure is the property value.

Using this scheme, CTOS defines a universal syntax-graph that will encode the syntax of every language possible within CTOS. It has only three kinds of syntax graph node: syntacticElement, syntacticLink, and syntacticProperty. Both syntacticElements and syntacticLinks have a linked list of syntacticProperty nodes attached.

All of the structure unique to a language is encoded in the properties and the srcHolders. There is a kind of srcHolder unique to each language, which is coded to understand the properties of that language. This means that grammar, the type system, and visualization reside inside the srcHolder.

To extend a language, then, is to modify the srcHolder for that language (the goal is to keep the translator unmodified). So the srcHolder is where grammar extensions, type system extensions, and visualization extensions will be defined: in the visualizer and modifier processors that reside inside the srcHolder.

The visualization extension is handled by the combination of the modifier, the visualizer, and the Display. Because there is no "text" form of languages in CTOS, to look at code, a visualizer generates a representation, which is in turn painted by a Display.

Visual representations are implemented by using vector graphics. The primitives of the language have pre-defined vector graphics to represent them, and custom syntax simply adds a new vector graphic drawn by the programmer (plus some drawing rules). Thus, arbitrary drawings created in packages like inkscape and Illustrator are used as custom notation.

There are some constants in visualizing a syntax-graph. That is, some visual cues remain the same for all custom notation: a command is indicated by some vector graphic; inputs are indicated by position relative to the command graphic; and output either semantically replaces the command plus inputs, or an arrow points from an output-position to an input-position.

Grammar extenstions are implemented inside the visualizer and modifier.

For source-modifying commands there are three kinds of visualization, depending on what one is looking at: viewing an end-user worksheet, viewing the system behind a worksheet, or viewing the specification to construct that system.

An end-user worksheet is a graphical view of a system of processors. The worksheet displays the contents of multiple Holders on the same worksheet, giving the feeling that they are all part of the same document. The worksheet also represents processors symbolically, indicating how data flows among the processors. Hence, one views source code mixed with processor symbols. The exact visualization is implementation dependent. For example, inputs to the system might be seen as variables, while srcHolder processors are seen as dashed boxes surrounding the source inside.

One interesting way of specifying communication to a processor might be the following. A processor is represented by the source used to create it. Free variables appear in that source. The same variable names are placed on the worksheet outside the processor. When the external variable is set to a new value, that value is communicated into the processor as an input. This allows a user to use relatively standard mathematical notation to indicate communication among processors.

Other processors might be displayed as some kind of symbol. An input would be indicated by a particular position relative to the symbol, as in "A + B" the A and B are taken to be inputs because of their position relative to the plus symbol. Or, an arrow might be used to indicate a data connection.

Regardless, the user-programmer only sees the worksheets. The extensions to EQNLang may cause new menu entries and other GUI gestures to become available to the user-programmer. These GUI extentions are used to modify source, and to modify the system.

A possibility of interest to those in the Physics community is for a symbol the programmer places on the worksheet to take a specified area as input and produce output source. The symbol is live so every time the input source becomes modified, the output source updates. Such output source may be specified as input to a translate-and-create symbol. So, the worksheet may have an equation entered by hand, which is inside the input box to a custom transform symbol, whose output goes into a translate-and-create symbol whose output in turn takes live variables on the worksheet as input. The live variables may be assigned to a file processor symbol or be assigned by hand. Either way, each time the variable's value changes, new output is produced by the processor created from the automatically modified equation. This is useful when one wishes to define a custom transform that takes equations as input, and then test that equations run through the transform produce expected numerical results.

In more detail, a programmer can use a transform that they create themselves that takes another equation as input, to produce a modification of that second equation, as is common in theoretical physics work. The result of transforming the original equation becomes a translated and running processor seen symbolically on the same worksheet. It efficiently computes data that may be graphed on the worksheet, allowing visual inspection. In this way, the physicist can tell quickly whether the original equation they tried gives expected results (currently, much of this process is done by hand using a math tool for some parts, pen and paper for others, and traditional programming languages for others).

Worksheet descriptions are a way to re-create a live worksheet. They are normally generated automatically and never seen by humans.

Systems are what are behind live worksheets. Some things are more easily specified by viewing the system directly and building it via GUI gestures. When viewing a system, processors appear as boxes.

## 2.2 Translatable Custom Notation

Translatable custom notation is translated by term re-writing. To define a custom translatable command, a re-writing rule is stated. During compilation, this re-writing rule is applied. After re-writing, the result is a combination of EQNLang primitives plus other custom translatable notation. As long as no cycles exist in the re-writing system, the resulting custom translatable notation brings the syntax one step closer to primitives-only. The translator repeats the term re-writing process until only EQNLang primitives remain.

The translatable type of custom notation is defined using four separate kinds of code:

1. specification of editor commands and what action the editor performs on the syntax tree for each
2. specification of the visual form of the notation, and how it corresponds to syntax tree elements
3. specification of how the GUI turns gestures into editor commands
4. re-write rules for the custom syntax-graph elements

Spec 2 is used by the visualizer. The visualizer uses this spec to turn syntax-tree elements into standard visual elements. Making the visualizer do this is straight forward because the syntax-graph form, the standard visual elements, and the visualizer have been designed together to facilitate it.

Given this, it can be seen that the first three kinds of spec form a cycle: editor-command -> syntax-tree mod -> visual-elements -> GUI display -> user gestures -> editor-command. The first kind of spec covers editor-command -> syntax-tree modification. The second kind covers (modified) sytax-tree -> visual-elements. The second kind also covers GUI display of the visual elements. This is because the second spec includes vector graphics that are given to the srcWindow, and the visualizer specifies the name of the vector graphic in a visual element. The user themself performs GUI display -> user gestures. The third kind of spec then covers user gestures -> editor-command, completing the cycle.

For the fourth kind of code, two cases exist: when the term-rewriting rule can be directly specified, and when the term re-writing rule must be generated. Some notation is so complex it requires a processor to generate the re-writing rule from the syntax graph (of custom notation) the user enters.

### 2.2.1 Direct translatable custom notation

For the majority of translatable custom notation the re-writing rule can be directly stated when the custom notation is defined. Figure 2 shows an example of defining the summation symbol, Figure 3 shows an example of using the summation symbol thus defined, and Figure 5 shows the result of applying the rewrite rule in Figure 2 to the use in Figure 3.

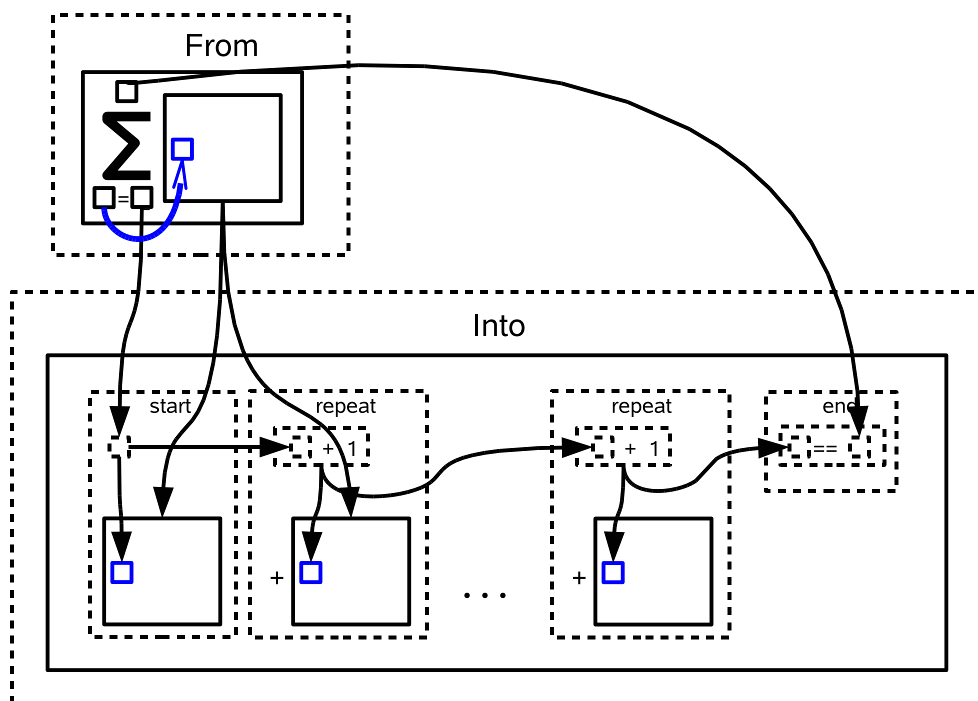


**Figure 2.** Defining custom notation for the summation symbol

$$\sum_{x=1}^{5} x \cdot x$$

**Figure 3.** Example of using custom notation

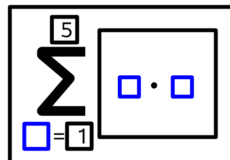


**Figure 4.** The resulting "From" box

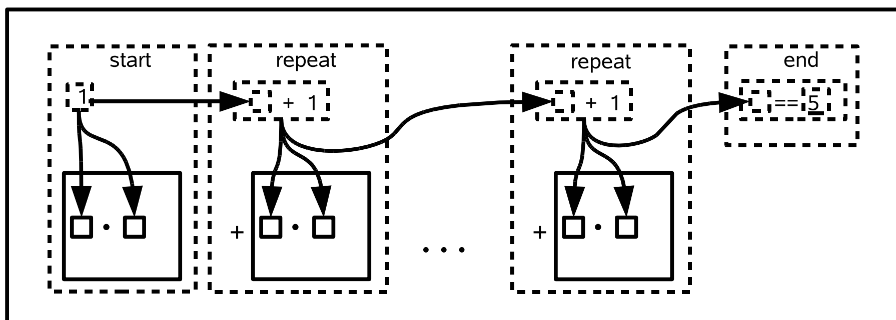


**Figure 5.** The result of applying the re-write rule defined in Figure 2 as used in Figure 3

In Figure 2, there appear a "from" dashed box and an "into" dashed box. The from-box holds the "$\sum$" vector graphic of the command, surrounded by empty boxes that represent the positions that inputs are placed. The blue line indicates that matches are searched for. When the command is used, as in Figure 3, something will be placed inside the box at the tail of the blue arrow, in this case "x". Something will also be placed inside the box that the head of the arrow is inside of, in this case "$x \cdot x$". When this re-writing rule is applied, a search will be performed to find all occurances of what's at the tail, inside the box the arrow-head is inside of, in this case, two separate matches are found.

In the into-box, at the center is an ellipsis. This is a primitive of EQNLang. The "start" box, the two "repeat" boxes and the "end" box are all part of the ellipsis notation. Ellipsis is the equivalent of a while-loop. It repeats the repeat-box until the evaluation in the end-box is true.

The arrows that go from the from-box to the into-box represent syntax that is copied. The arrow starting at the large empty box points to an identical large empty box, indicating that the syntax is taken from that position in the from-box and placed into the into-box. Figure 4 shows what the from-box looks like for the usage seen in Figure 3. Figure 5 shows the result of performing the re-write. It can be seen that what instantiated, from the use, into the big empty box has been copied directly into the final ellipsis. Only the color has changed.

The blue in the from-into indicated that a repeat was done during copy for every match found. So the blue only had meaning inside the re-write operation. Once the copy of syntax from the from-box to the into-box completes, the blue color is out of scope.

It can also be seen in Figure 5 that the "1" and the "5" have been copied down, according to the arrows between the from-box and the into-box.

The dashed boxes are part of the ellipsis notation. One may use them to write a kind of shadow-code that calculates something for each position of the ellipsis. The dashing indicates that, semantically, all of the dashed-box operations are performed before any of the sold-box operations. In this example, all five repetitions will be generated before any actual calculation begins. In each repetition, the results of the dashed-box calculation will be substituted down into the solid boxes.

However, these five repetitions will never actually be created anywhere. The translator simply knows that the semantics of ellipsis mean that they could be generated. An advantage of these semantics is that dependencies are explicitly stated. Also, only a single primitive is needed: FOR loops, DoAll loops, While loops, do-while loops, and so forth all can be defined as re-write rules that map onto ellipsis.

### 2.2.2 Generated Re-Write Rule Type of Translatable Custom Notation

Sometimes the notation includes complex rules for determining the action to take. For example, in Tensor notation, abstract indexes are used as a sort of semaphore. One places a sequence of characters after the TensRep name, each in either super-script position or sub-script position. The particular character used has no significance, only the position in the sequence and raised or lowered matters, and which other postions have the same character. The operation one performs on the TensRep changes depending on which positions match. The operation to perform determines the re-writing rule. To determine the re-write rule, one must first parse the index postions and match indexes of the same character. Then one can generate a rule based on where the matching indexes occur (and which index was raised and which lowered).

For this kind of custom notation, processors must perform work during editing of the code. The environment has provided a generic facility for this. At the time that a srcHolder is added to a programSpace, part of the add-process is a conversation between the programSpace and the srcHolder. During this, the srcHolder can point the programSpace to a language-specific architecture description. The programSpace will generate inside itself whatever system the ADL (architecture description language) specifies.

EQNLang uses this facility to instantiate a “re-write-rule factory-factory”. This factory-factory has two responsibilities: to create factories for specific custom notation, and to dispatch requests to the appropriate created factory.

SrcHolders in which generated-rewrite-rule custom notation is entered collect the notation’s syntax sub-graph from the user, via the GUI. When the sub-graph is complete, the srcHolder sends the sub-graph to the re-write-rule factory-factory, asking to receive back the name of the generated re-write rule.

When the re-write-rule factory-factory receives a syntax sub-graph, it looks in the root node at the name of the custom-command. It then looks to see if it has created a factory for this kind of custom-notation yet. If not, it searches for and finds the application code that implements a re-write rule factory for that kind of custom-command. It uses the environment’s name-discovery service to find the srcHolder holding the re-write-rule factory’s code, and uses the creator to create a processor from it.

Next, the re-write-rule factory-factory sends the syntax sub-graph to the factory for that kind of notation. That factory processes the syntax sub-graph, producing the syntax for the re-write rule, and giving the rule a name. It hands the re-write rule to the programSpace, asking it to put the syntax into a new srcHolder. The factory also hands the rule-name to the factory-factory, which hands the name to the original requesting srcHolder. The editor in that srcHolder puts the name into its syntax-graph. Now, the syntax-graph the editor has looks exactly like the syntax-graph for custom notation with re-write rules entered by hand. The only difference is instead of the srcHolder holding the re-write rule being filled by hand by a programmer, the srcHolder was instead filled by a factory.

When an application programmer writes a re-write-rule factory, they have the full EQNLang available. The factory is a processor, like any other. It simply takes syntax sub-graphs as input and produces syntax for re-write-rules as output.

## 2.3 Source Manipulating Custom Notation

EQNLang, as its name implies, is meant to facilitate math, especially math of the kind encountered in theoretical physics work. When writing a proof, for example, from one line to the next some transform has been applied. EQNLang supports this kind of work by allowing a symbol to be placed on a (live) worksheet that takes the line above as input, applies the transform indicated by the symbol, and automatically produces the line below. The user may then place another symbol indicating another transform, and so on. If no symbol exists for the transform the user desires, they create a new symbol and define the transform that goes with it.

Custom source-manipulating notation is used in three places: live worksheets, worksheet descriptions, and systems.

### 2.3.1 Live worksheets

On live worksheets, the programmer is interacting with a view of equations, and/or code. They enter equations, manipulate them, then perhaps run some numbers through the final equation to verify that it gives accurate predictions.

In this process, the programmer might perform some GUI gesture to cut selected equations and replace them with a transformed version. This is referred to as an edit-time use of the transform. The other way a programmer might use a source-modifying transform is to place a symbol on the worksheet that represents the transform. A selection of equations on the worksheet is connected as input to the symbol, and the output appears below the symbol on the worksheet. Any time the input changes, the transform is re-run and produces new equations that appear on the worksheet.

For example, one may wish to apply the Fourier transform symbolically. The input equation is substituted into the definition of the Fourier transform. This yields an equation as the result. To do this on a live worksheet, one selects the input equation, places the Fourier transform symbol, and the resulting equation pops out below it.

One may now wish to use the resulting equation the same way one uses normal source. The only complication is how to handle connections. In the first version of EQNLang, this is done by using variable names. Variable names are treated as wires. Variables that appear on the worksheet earlier than an equation containing the same names, are treated as inputs to the equation. Thus, if the result of source manipulation contains variable names in input positions, and those same variable names appear earlier in the worksheet, then the earlier assignments act as inputs to the manipulation results. The variable names in input positions will receive whatever was the most recent assignment to that variable name.

On a live worksheet, earlier and later are determined by scan-order. Starting in the upper left corner, a raster-scan is made that travels along a line from left to right, then goes to the left hand edge of the next line and repeats. Things that are encountered earlier in this scan are earlier in the worksheet. Those that happen later are later in the worksheet.

So, variables that are assigned-to inside source-manipulation results cause the assigned value to be available to any uses that appear later in the worksheet. And vice versa.

For example, one may place a data-stream-source symbol on the worksheet. It can then be assigned to a vaiable. If one defines input to a manipulation to have that variable, the the result of manipulation will also have the variable. The variable's value will thus transmit from the data-stream-source into the manipulation result. Now, to see the numeric result, the manipulation result assigns to a variable. The same variable name is connected to a graph symbol. Data thus flows from the data-stream-source, into variable A, from A into the manipulated equation, which assigns to variable B. The numeric results flow through B into the graph and appear as a plot for the user to inspect.

### 2.3.2 Worksheet Descriptions

Worksheet descriptions are normally generated from live worksheets: one creates the live worksheet by hand then causes its description to be generated. The description can be used to recreate the live worksheet, later or in a different OSInstance.

Whether generated or entered by hand, a worksheet description is a form of architecture description language for a worksheet plus the system behind the worksheet. The system behind the worksheet is specified with normal the normal ADL (architecture description language). The worksheet itself is simply another entry in the ADL to create the worksheet processor, plus an entry to cause a srcHolder holding the syntax-graph that was on the live worksheet to be sent to the newly created worksheet to repopulate it.

### 2.3.3 Used in a system

A srcManipulator can be placed within a system of processors in four ways:

1. by directly creating and connecting it in a live system using the GUI
2. by using the GUI to instantiate a creator then connect the programSpace holding the srcManipulator code to the creator and manually connect the input and output of the creator-generated srcManipulator to other processors
3. by using architecture description primitives to state the creation of a srcManipulator and the connections to the resulting processor
4. by using architecture description primitives to state the placement of a creator, the placement of a programSpace holding the srcManipulator code, the connection of the programSpace to feed the creator, and the placement and connections of the creator-generated srcManipulator processor.

In the first two cases, one is working on a live system by means of a GUI or other command interface. In the last two cases, one is working with an architecture description language that states how to build a live system.

### 2.3.4 SrcManipulators in Live Systems

Such a system lives inside a workSpace, which performs checks on processor interfaces against the attempted connections of those processors, as well as other system tasks.

In case 1, one uses GUI gestures to create a srcManipulator. In this case, the programSpace holding the source code of the srcManipulator does not appear in the workSpace. It is only specified by GUI gestures during creation of the srcManipulator. The resulting srcManipulator appears in the workSpace as a processor-box. One uses further GUI gestures to connect that box to other processors in the system. The source code of the srcManipulator contains an interface that determines what connections are allowed. One may send one or more syntax-trees and/or commands and parameters to the srcManipulator, via connections made to it.

In case 2, one first uses GUI gestures to instantiate a programSpace holding the code for a srcManipulator, then to instantiate a creator, and connect the two. When one placed the creator, a black-box popped up as its output. One then uses further GUI gestures to connect that black-box to other processors in the workSpace. One must instantiate (create) the programSpace and connect it to the creator before the workSpace allows making any connections to the black-box. This allows the interface specified in the code in the programSpace to be checked against the connections made to the black-box. Subsequent changes made to the interface in the source will be checked by the workSpace.

### 2.3.5 SrcManipulators in Architecture Descriptions

Architecture descriptions of systems are normally generated by tools: one builds the live system, then has the architecture description of it generated. With the resulting description, the live system can be re-created, at a later time or inside other OSInstances.

Two approaches can be taken to generating systems from architecture descriptions: make ADL the language of the creator, or make ADL be commands to workSpaces. Each has its attractions.

When going with making ADL the language of the creator, the creator processor implements the ADL interface the same way a Pentium implements the x86 ISA. To create a live system from an architecture description, one gives it to the creator. The creator builds the system by implementing the architecture description primitives.

In this case, the only primitives needed in the ADL are for:

- finding the names of srcHolders
- causing creation of a processor from code inside a srcHolder, and placement into another processor
- creating connections among processors
- causing an existing processor to be placed, by “tunneling” into a processor
- causing the creator processor, along with its created processor, to be placed into a processor

From these five, every possible system can be constructed. However, the OSInstance must already contain srcHolders holding the source to create standard types of processor such as workSpaces, programSpaces, srcWindows, new and empty srcHolders, and so on. In addition, if one wishes to create any application-defined processors, one must have the source code for those processors in some srcHolder in the OSInstance before giving the architecture description to the creator.

If, on the otherhand, one goes with ADL as commands to a workSpace, one creates the top-level workSpace by hand, then gives that workSpace the ADL. The ADL will be heirarchically organized. The top level of the ADL will be commands that create and place processors into the top level workSpace. Some ADL commands will specify creating a new, embedded, workSpace and then handing a sub-tree of the ADL to the embedded workSpace. Thus, a system builds itself.

The only primitives needed in addition to those for the creator-case are:

- a primitive that demarcates an ADL sub-tree (defines the sub-tree’s boundaries)

- a primitive that causes an ADL sub-tree to be handed to a processor

This is probably the more flexible option, allowing customization of ADL by writing new kinds of Space processors to give the new custom commands to. However, it may have logistical implications for the proliferation of Space types and compatibilities. Both paths look interesting.

Either way one goes, no special syntactic mechanism is needed for srcManipulators in architecture descriptions. In live worksheets, srcManipulators had to be treated specially because they implied modifications to the system "backing" the worksheet. But in ADL code, one is specifying the system directly, so a srcManipulator is just another processor.

When considering custom notation in architecture descriptions, the situation is a bit simplified because architecture descriptions are, by their nature, never inter-mixed with live processors. So one can always use translatable custom notation in architecture descriptions, even when specifying the creation and interconnection of srcManipulators. Custom ADL notation, even involving srcManipulators, can be specified to re-write itself into the above primitives (or primitives plus other custom ADL notation).

Only one compilication exists, which is checking interfaces when creating connections between processors. The source code of the srcManipulator is not necessarily known at the time of the system creation. Nor is the syntax-graph that will flow into the srcManipulator necessarily known. Thus, no analysis can be done to discover the types of the ports on the processors created from the output of the srcManipulator. The creator must connect the creator-generated processors blindly. This means that run-time checks will have to be performed. These checks can be done on code when it is inside the creator, or the workSpace can perform checks on the ports of a creator-generated processor before disconnecting the old creator-generated processor and connecting up the new one.

Who checks the interfaces of processors generated from srcManipulators is independent of ADL choice. After the system creation is complete, the code that leaves a srcManipulator is put through a creator. So it only matters that the resulting processor's interface is checked every time the srcManipulator outputs new code.