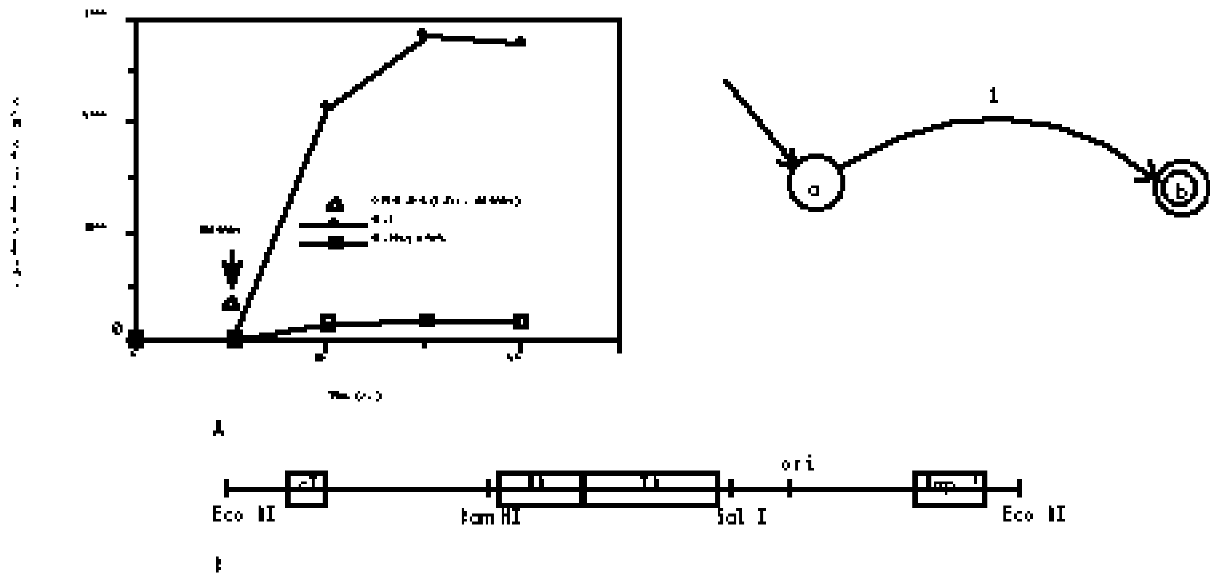
The Diagram Understanding System - Strategies and Results[1]

Technical Report by R. P. Futrelle - May 2007

Biological Knowledge Laboratory (BKL)
College of Computer and Information Science WVH202
Northeastern University, 360 Huntington Ave.
Boston, MA 02115



Abstract

Strategies for syntactic parsing of parsing diagrams are explained. The basic strategy is the use of *Context-based Constraint Grammars* to express and guide the parsing process, as well as *spatial indexing* that allows the system to evaluate spatial constraint predicates rapidly. It is notable that the system described here was complete and operational in 1996, which made it possibly the first such system to fully parse a variety of actual diagrams drawn from the research literature, notably x,y data graphs and linear gene diagrams, as well as finite-state automata diagrams created for the project. See: Futrelle, R. P., & Nikolakis, N. (1995). *Efficient Analysis of Complex Diagrams using Constraint-Based Parsing*. In ICDAR-95. Montreal, Canada, 782-790.

---

[1] This technical report is based on the *The Diagram Understanding System Demonstration Site* http://www.ccs.neu.edu/home/futrelle/diagrams/demo-10-98/
It is available online, along with a number of other papers on the diagram research in the BKL, at: http://www.ccs.neu.edu/home/futrelle/papers/diagrams/TwelveDiagramPapersFutrelle1205.html

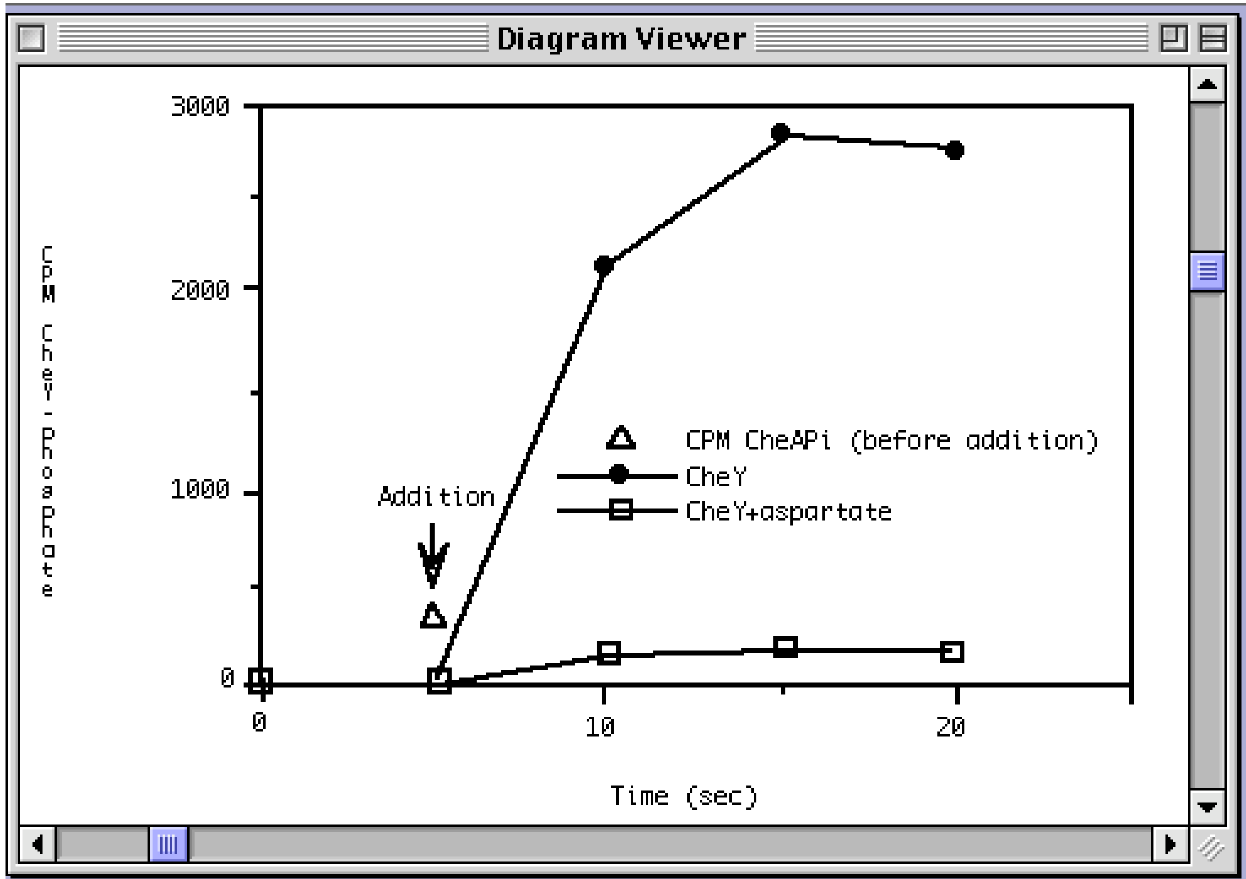
Contents

* Visualizing the Relations between Objects and Images
* Using the Lisp Inspector
* The Structure of Grammars for Diagram Parsing
* How the Grammar Rules Drive the Parsing Process
* Parsing runs with timing
* Object Sharing in parsed structures
* Spatial indexing and geometrical computations
* Parsing finite-state automata diagrams and gene diagrams
    o Finite-state automata diagrams -- using sharing for context
    o Gene diagrams

Appendix A - Complete copy of a grammar for x,y data graphs
Appendix B - Full finite-state automata grammar
Appendix C - Full gene diagram grammar
Appendix D - Relation of this to ideas/systems in FRVDR98

Introduction

The original set of web pages on which this report is based was prepared for the demonstration of the Diagram Understanding System (DUS) at the Fall AAAI Symposium on Formal Reasoning with Visual & Diagrammatic Representations (FRVDR98), Orlando, Florida, October 1998. They represented work done by Futrelle and various graduate students over a number of years at Northeastern University, Boston. The overall goal of the Biological Knowledge Laboratory is to develop systems that can discover and exploit the content of scientific documents, e.g., using AI techniques for representing the conceptual structure of the documents. Biological research papers in particular contain numerous figures, so discovering figure content is important. The discussion here focuses on diagrams, by which we typically mean figures that are made up of vector elements: lines, polygons, text, etc., as opposed to continuous-tone images, e.g., photographs. The three thumbnails on the title page of this report represent the classes of diagrams that we initially focused on. We explain how our system works by presenting a series of examples of its use to parse diagrams and to view the structure of the parses. We also discuss the nature of the grammars we use. All screen shots and example runs in this demonstration were produced on a Macintosh G3 Series PowerBook, 300MHz, using Macintosh Common Lisp, allocated 20MB of space. The typical parse times for diagrams quoted below, a few seconds, would be in the hundreds of milliseconds on current machines (2007).
We have concentrated on diagrams taken directly from the published literature, rather than simple, stylized diagrams that we create. The diagrams consist of anywhere from 50 to 1000 primitives (lines, polygons, text, circles, and Bezier curves). All diagrams in this demonstration were redrawn by us from the printed originals to produce the vector form needed by our system. Here is a such a diagram (Borkovich and Simon, *Cell*, **63**, 1990, pg. 1343), as shown in the DUS Diagram Viewer.



A grammar that defines the structure of such an x,y data graph (a Context-Based Constraint Grammar) is used to analyze (parse) the diagram. The parse is a collection of interrelated graphical objects, from primitives such as lines or Bezier curves, to higher-level objects such as X-Axis or Data-Cluster. Parsing proceeds by:

First - Loading a grammar using defgrammar, and installing all objects in the source image into the spatially associative array pyramid structure (SPAS == Spatially Associative Substrate), in either order.
Second - Parsing the installed image using (parse <node>) where the <node> chosen is usually the top-node, e.g., Image in our x,y data graph example.
The diagram viewing system, DUSI (DUS Inspector) is used to examine the relation between the solution objects and the corresponding visually observable structures. All the figures in this set of web pages were obtained as screen shots from DUSI.

Relation of this work to ideas/systems discussed in FRVDR98

This work is relevant to formal reasoning about diagrams in at least the following ways:

* Parsing diagrams is reasoning about diagrams, in that it is a constructive proof that the parsed entity is an instance of a certain description of a class of objects.

* To do automated reasoning about a diagram, it is often useful to start with a structured description of the diagram, rather than a collection of primitive objects (lines, polygons, etc.). A parse produces such a description.

* Building a framework for writing grammars raises important issues about how to specify the geometrical relations that should hold within a diagram of a certain class.

* Designing a computational strategy for parsing a diagram forces us to think about how to discover and keep track of geometric relations within a diagram in an efficient way.

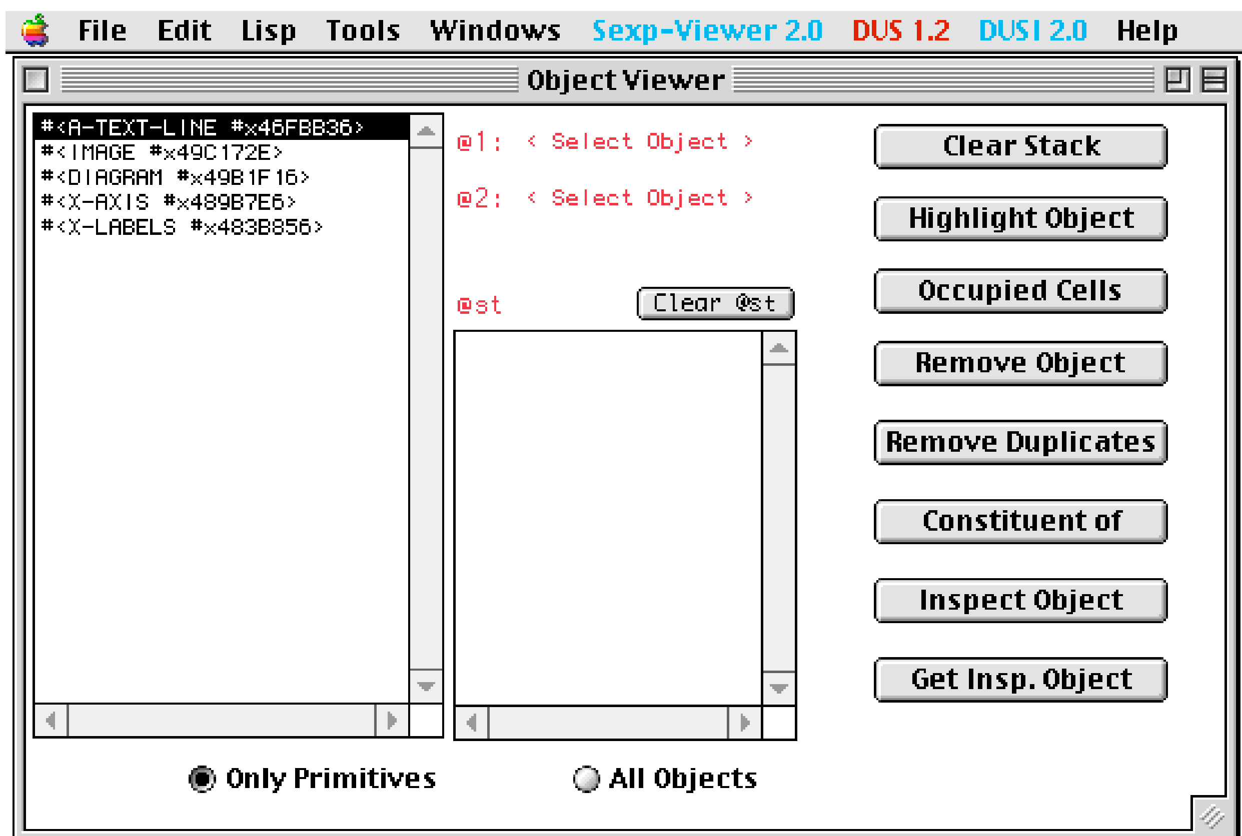
* Any approach to parsing helps to elucidate problems such as ambiguity in diagrams and their descriptions.

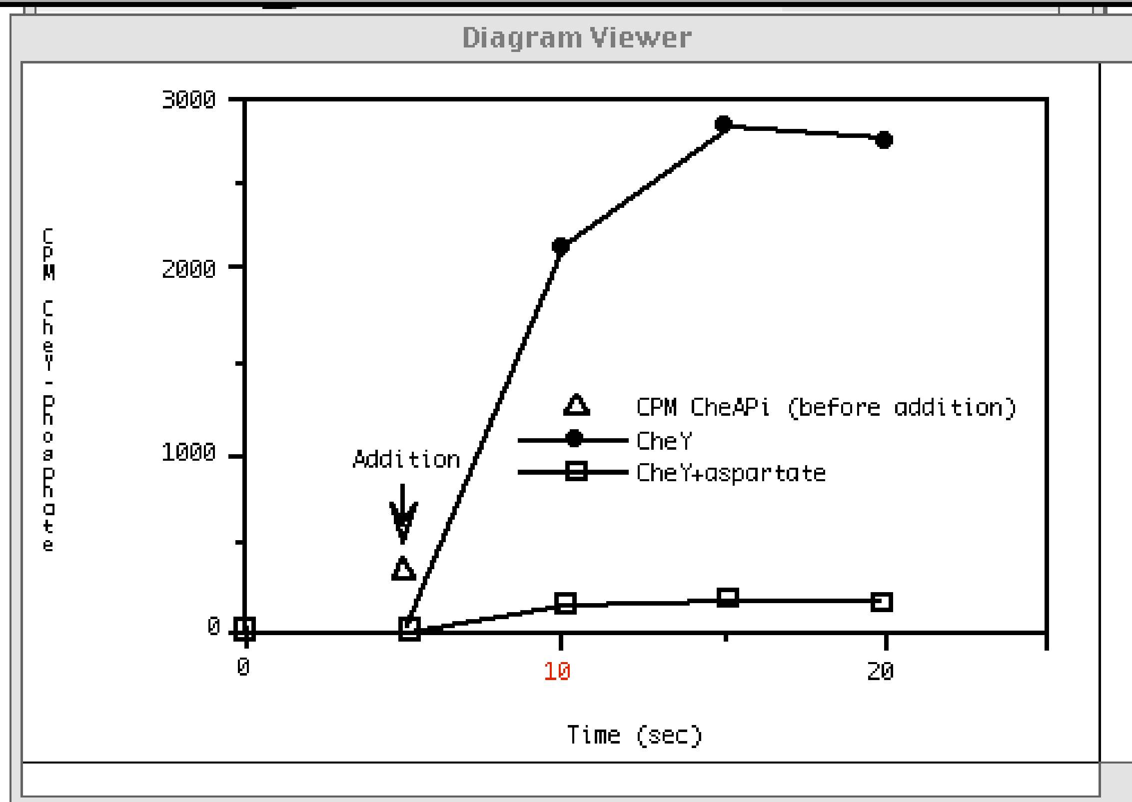
Viewing the relations between objects and images

Each object in the parsed diagram, whether a primitive such as a line or a higher-level object such as X-Axis, is represented by a program object at run time, specifically a CLOS class instance in Lisp. To visualize and examine the correspondence between the run-time objects and their counterparts in the visual image, we developed the DUS Inspector, or DUSI. On the next page, we show the Object Viewer (OV) and the Diagram Viewer (DV) in DUSI.

The need to simultaneously view the program/parse objects and the corresponding graphics cannot be over-emphasized. In a natural language parse of a sentence, the resulting tree structure typically ends in a set of ordered leaves that are in close to one-to-one correspondence with the words in the sentence.

In the figure on the next page, the x-axis tick label, "10" has been selected by clicking on it in the DV. The corresponding object, "#<A-TEXT-LINE #x46FBB36> appears selected at the top of the stack window in the OV. In addition, with the text-line item selected, the "Constituent of" button in the OV was pressed, causing all ancestors of the text-line in the parse tree to be placed on the stack. This shows that A-TEXT-LINE is a constituent of an X-LABELS object which is in turn a constituent of an X-AXIS object which is a constituent of a DIAGRAM object. The DIAGRAM object is the only constituent of the overall IMAGE object. (A multi-part plot, a set of x,y diagrams, can have a set of more than one DIAGRAM in an IMAGE.)





The Object Viewer (OV) and the Diagram Viewer (DV) in DUSI (the DUS Inspector).

Using the Lisp Inspector with DUSI

If the X-AXIS object in the OV is selected, various things can be done with it. On the next page, we show the results that follow from pressing the Highlight Object and Inspect Object buttons.

Pressing the Inspect Object button in the OV causes a Lisp Inspector window to open, inspecting whatever object is highlighted in the OV stack. Pressing the Highlight Object button causes all the primitive objects which are constituents of the highlighted stack item to be highlighted in color (in red) in the DV window.

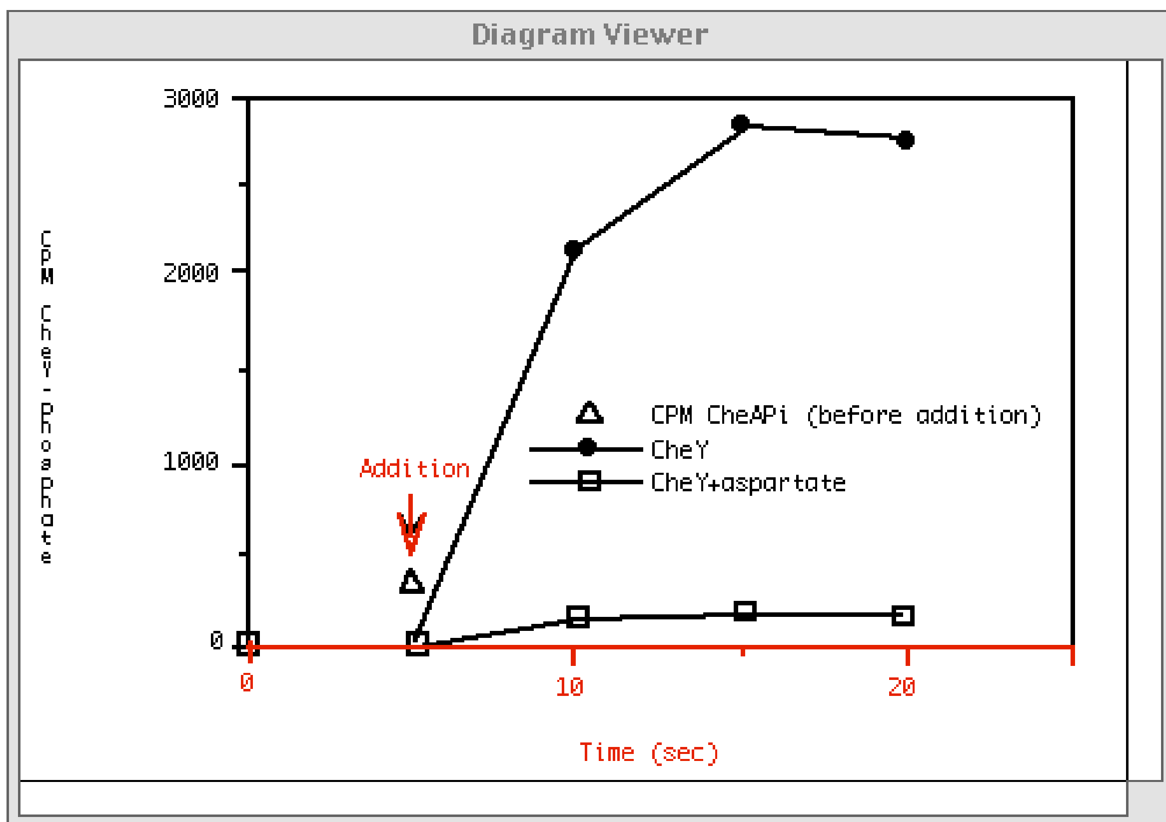
In the bottom DV window, the components that make up the X-AXIS object are highlighted. These include the axis line, the axis label, "Time (sec)", the tick marks and their numerical labels, and the annotation text and arrow. The contents of the X-AXIS CLOS object are shown in the Lisp Inspector window above the DV. The object has many slots, but the primary ones of interest here are the last five X-AXIS-LINE through X-TEXT, which are the constituents of the X-AXIS production in the grammar.

File Edit Lisp Tools Windows Sexp-Viewer 2.0 DUS 1.2 DUSI 2.0 Help

#<X-AXIS #x489B7E6>

Commands | Edit Value | Resample

```
#<X-AXIS #x489B7E6>
Class: #<STANDARD-CLASS X-AXIS>
Wrapper: #<CCL::CLASS-WRAPPER X-AXIS #x420D1CE>
Instance slots
INFO: #<Unbound>
SYSTAG: #:G58283
TAG: #:G57805
LEVEL-ARRAY: #(#<G-O-REGIONS #x489C126> #<G-O-REGIONS #x489C16E> #<G-O-REGIONS
LEVEL-X-ARRAY: #(#<G-O-REGIONS #x489C13E> #<G-O-REGIONS #x489C186> #<G-O-REGIO
LEVEL-Y-ARRAY: #(#<G-O-REGIONS #x489C156> #<G-O-REGIONS #x489C19E> #<G-O-REGIO
BB: #<B-BOX #x48A3146>
ASPECTS: (#<ASPECT #x4750726>)
STATS: NIL
CONST-OBS: #<A-SET #x489C10E>
LEAF-OBS: #<A-SET #x48A3DB6>
DESCR: #<Unbound>
X-AXIS-LINE: #<X-LINE #x480E866>
X-TICKS: #<OWN-X-TICKS #x482EACE>
X-LABELS: #<X-LABELS #x483B856>
X-ANNOTATION: #<X-ANNOTATION #x48913B6>
X-TEXT: #<X-TEXT #x48996FE>
```



Showing the results that follow from pressing the *Highlight Object* and *Inspect Object* buttons in the Object Viewer (top) and the Diagram Viewer (bottom).

Introduction to grammars -- A fragment defining the X-Axis of a data graph

Below we briefly discuss a fragment of the full x,y graph grammar dealing with the X-Axis structure highlighted in the previous example. The fragment consists of three productions. The full grammar is also available for browsing. After looking over this fragment, you can proceed to the discussion of how the grammar rules drive the parsing process.

The overall structure of this grammar fragment is:

* An Image is defined as a set of diagrams so that a multi-part diagram can be parsed, e.g., two data-graphs, one above the other.

* A Diagram is made up primarily of axes and data.

* An X-Axis has five components:

  o The X-Axis-Line which is the long line to which the tick marks are attached.
  o The X-Ticks which are a set of short vertical lines very near the x-axis-line.
  o The X-Labels which must each be close to and below their corresponding ticks.
  o X-Annotation which is an arrow with attached text.
  o X-Text which is the overall axis label below the tick labels.

The grammar fragment:

```
;;; ***************** < Image > *****************

   ( Image -> Set ( Diagram ));

;;; ***************** < Diagram > *****************

  ( Diagram -> Axis  X-Axis  Y-Axis  Data
      (Axis)
      (X-Axis   ($ :axis Axis))
      (Y-Axis   ($ :axis Axis))
      (Data
       ($ (difference*  (contain Axis '?)
                        (union* X-Axis Y-Axis))
          :x-ln (ln (X-Line Axis))
          :y-ln (ln (Y-Line Axis))
          :axis Axis)));
```

```
;;; *****************  < X-AXIS >  *****************

    ( X-Axis -> X-Axis-Line   X-Ticks   X-Labels  X-Annotation   X-Text
       (:optional X-Annotation X-Text)
       (X-Axis-Line  (X-Line (get-val axis)))
       (X-Ticks      ($ :x-line X-Axis-Line)
              :constraints (>= (size X-Ticks) 3))
       (X-Labels     (below '? X-Axis-Line :strip t))
       (X-Annotation (difference* (near X-Axis-Line 700)
              (union* X-Ticks X-Labels))) ; label-size
       (X-Text
              (near&below '? X-Labels (* 2 (height X-Labels)))))
```

Parsing details for a simple grammar

We will examine how parsing proceeds for a grammar that only describes the y-axis line and its tick marks and does so in a very simple way. When a grammar is entered into the system, a Lisp defgrammar macro expands the form into one that includes an additional defrule macro, applied to each rule structure. When the defrule forms expand, they define the CLOS classes for each non-terminal as well as the goal predicates for the search and the generator functions needed. These macros act as the parser generators. Once they have been applied, the parsing process then acts as a top-down tree-search.

A simple grammar with three rules:

```
;;; ***************** < X-Ticks > **************

( Y-Ticks -> Ticks Y-Line
       (Y-Line)
       (Ticks (touch Y-Line '?) :constraints (> (size Ticks) 2)))

;;; ***************** < X-Line > **************

   ( Y-Line -> Line
        (:constraints (vertp Line)
        (long Line)))

;;; ***************** < Ticks > **************

   ( Ticks -> Set(Line)
        (:element-constraints (horizp Line)
                              (short Line)))
```

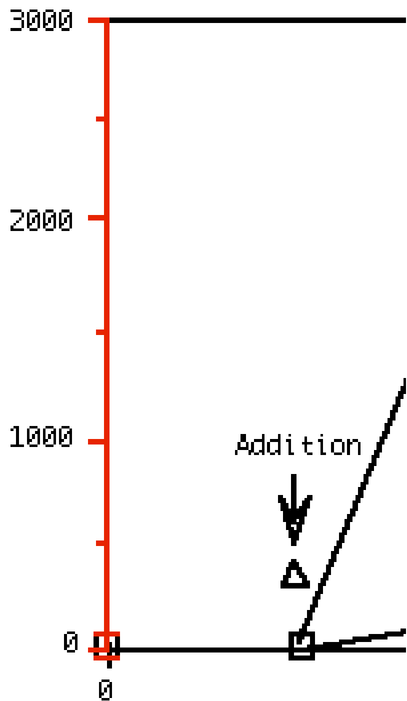
The parse is assumed to be started at the top node, Y-Ticks. Parsing proceeds by a depth-first search to discover and build elements that satisfy the various geometric constraints. The order of the search is dictated by the order of the constituents in the body of the rule (not in the order of the constituents in the "->" production).

1. In the Y-Ticks rule, the Y-Line constituent is searched for first. There are no specific restrictions on the Y-Line search included in this Y-Ticks rule

2. In the Y-Line rule the Line primitive is the only constituent, and it must obey the constraints that it be vertical and long.

3. Returning to the Y-Ticks rule, Ticks are searched for next. The expression "(touch Y-Line '?)" is a context generator. The set of all elements in the diagram that obey this constraint is created and the Ticks constituent rule is then entered with this context restriction -- Ticks members can only be chosen from the set of elements in the diagram that touch the long vertical line found by the Y-Line rule. The context is propagated to a constituent rule as an inherited attribute.

4. The Ticks rule is then entered. Ticks is a set of Line primitives. The elements of the set are chosen from the context inherited from the Y-Ticks rule, as explained in step 3. The elements of this set are further filtered by constraining every one of them to be both short and horizontal. Any subset of the y tick marks would satisfy this rule. But our algorithm attempts to choose the maximal set of elements that satisfy the rule. In this way, a single rule can return twenty tick marks or a hundred data points in one step. In the Y-Ticks rule, an acceptable set of ticks must have greater than 2 members, the constraint ":constraints (> (size Ticks) 2))".



Our grammars, because of the way they are designed and operate, are called Context-based Constraint Grammars. It is essentially an Attributed Multiset Grammar, but its use of context is new. A rule inherits a context specified in its parent that restricts the set of items within which the rule is allowed to search for a solution. This successive narrowing of the search set helps to make the parsing process more efficient. Parsing generates a set of all distinct solutions that satisfy the constraints. In the example above, the set is a singleton. In addition, the same element can be reused or shared by various parts of a single solution, see the discussion of sharing.

When the parse is complete we can highlight the elements in the Diagram Viewer that make up the Y-Ticks structure, as shown on the right. Note that this simple grammar also identifies the top and bottom sides of a data point as tick marks. This is avoided in our more complex grammars by requiring that a tick mark not be a line that participates in a polygon.

Parsing x,y data graphs -- Sample runs with timing

As we said earlier, the parsing run-times listed below would easily be 10 to 20 times shorter using a contemporary system (c. 2007). The first parsing example is based on the simple x,y data graph presented in the Introduction. Parsing proceeds in two steps:

* Loading a grammar using defgrammar, and installing all objects in the source image into the spatially associative array pyramid structure (SPAS == Spatially Associative Substrate), in either order.

* Parsing the installed image using (parse <node>) where the <node> chosen is usually the top-node, e.g., 'Image' in our x,y data graph example.

The runs below were done on a Macintosh PowerBook G3 Series, 300MHz machine with 20MB allocated for the Lisp image.

Here is a trace of loading the grammar, which takes about 8 seconds (loading is needed only once, for parsing any number of diagrams covered by the grammar):

```
? (time (load-grammar))
(LOAD-GRAMMAR) took 8,021 milliseconds (8.021 seconds) to run.
Of that, 72 milliseconds (0.072 seconds) were spent
in The Cooperative Multitasking Experience.
 3,558,448 bytes of memory allocated.
#P"Macintosh HD:Research:DUS-comps3:Demo98:Grammars:x-y-grammar-final.lisp
```

Then the image objects in the diagram is installed, taking about 1.3 seconds:

```
(INSTALL-IMAGE FILE) took 1,309 milliseconds (1.309 seconds) to run.
Of that, 39 milliseconds (0.039 seconds) were spent
in The Cooperative Multitasking Experience.
 3,499,408 bytes of memory allocated.
```

And then the parse itself is performed which takes just under 1 second:

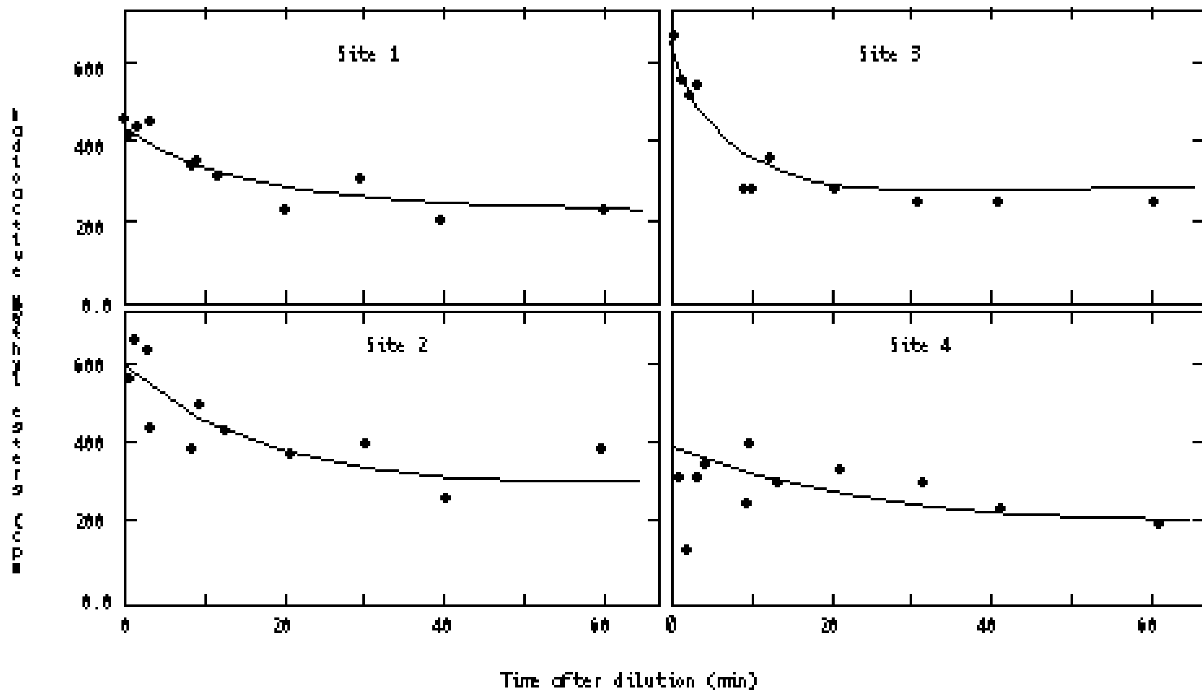
```
(SETQ *SOLUTION* (PARSE TOP-NODE)) took 943 milliseconds (0.943 seconds) to
run.
Of that, 12 milliseconds (0.012 seconds) were spent
in The Cooperative Multitasking Experience.
 1,920,504 bytes of memory allocated.
#<INSPECTOR-WINDOW "(#<IMAGE #x30DD006>)" #x30E86A6>
```

A larger and more complex data graph

Here are similar timing results for the more complex multi-part data graph shown below (Terwilliger, T.C., Wang, J.Y., and Koshland, D. E., Jr. (1986a) *J. Biol. Chem.* **261**(3), 10814-20, Fig. 3, pg. 10187). This diagram was made up of N=146 primitives, yet it took less than three seconds to parse, after an installation taking about nine seconds. Many critical parts of the analysis such as discovering data point sets and tick mark sets run in linear time, which is a major reason why such large N values can be handled efficiently. (Nikolakis, N., *Diagram Analysis using Equivalence and Constraints* (PhD dissertation), in College of Computer Science. 1996, Northeastern University: Boston, MA. 198 pgs)

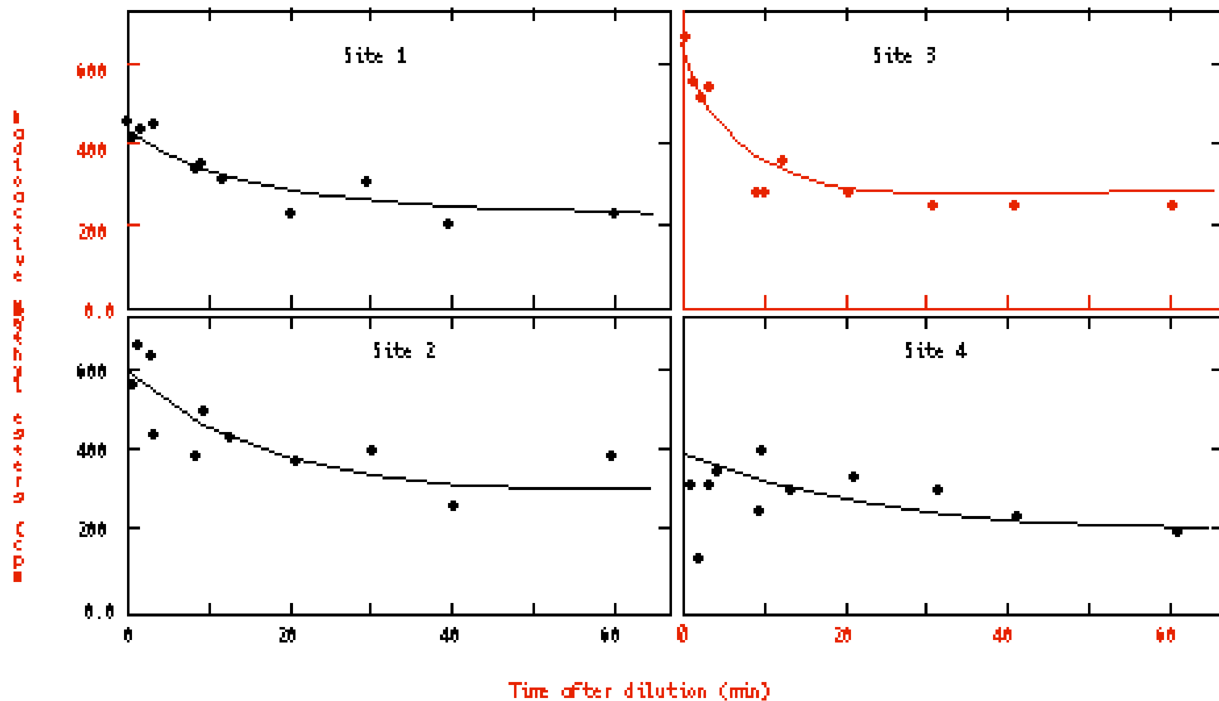
```
(INSTALL-IMAGE FILE) took 8,912 milliseconds (8.912 seconds) to run.
Of that, 2,297 milliseconds (2.297 seconds) were spent
in The Cooperative Multitasking Experience.
860 milliseconds (0.860 seconds) was spent in GC.
 10,995,456 bytes of memory allocated.

(SETQ *SOLUTION* (PARSE TOP-NODE)) took 2,404 milliseconds (2.404 seconds) to
run.
Of that, 236 milliseconds (0.236 seconds) were spent
in The Cooperative Multitasking Experience.
 4,973,040 bytes of memory allocated.
```



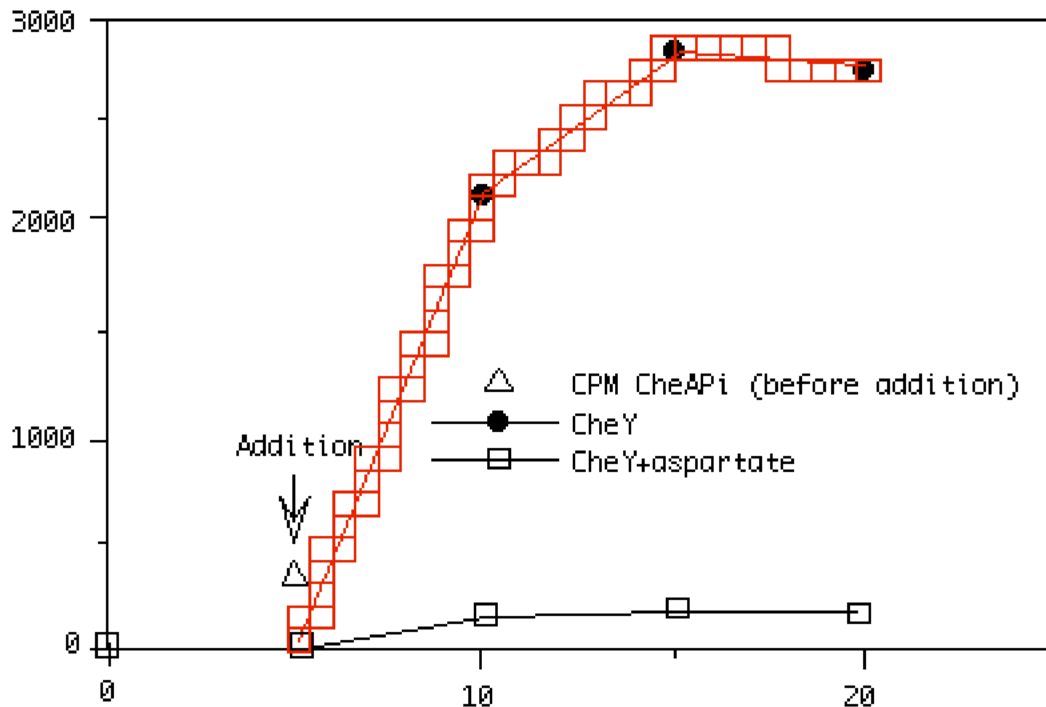
Objects in the image can be members of multiple parsed structures

This parsing example is based on a four-part x,y data graph. When the parsed elements for the upper-right data graph, "Site 3", are highlighted, we can see that various components below and to the left, but quite distant from the data itself, are part of the solution. The y-axis label on the far left is shared by all four of the parses for "Site 1" through "Site 4, that is, it appears as a constituent in all four of these components of the overall image parse. The numerical labels and tick marks on the upper left are shared by "Site 1" and "Site 3", while the numerical labels on the lower right are shared by "Site 3" and "Site 4". The x-axis label at the very bottom is shared by all four data graph parse components. Sharing is common in graphics, but is not allowed in many approaches to parsing. The "remote" components that end up being shared are found by using X-Or rules that first look for nearby elements, but if they cannot be found, remote but properly aligned elements are searched for (see the X-or construct in the X-Ticks rule in the complete x,y grammar).



Objects are installed in cells in a spatial index

A spatial index (SPAS == Spatially Associative Substrate) is used to store the regions occupied by the primitives and higher-level objects in a diagram. The basic collection of cells is typically a 64x64 square array, covering the space occupied by the diagram. Each object is installed in the array by creating an object reference in every array cell that the object occupies or passes through. The spatial array operates as an associative index mapping from 2-space to objects. In addition, two linear spatial projection arrays for the x and the y axes are also filled in the same way. The spatial array is used to efficiently compute spatial relations that are important in parsing. For example, finding out whether two objects A and B are near one another is done by inspecting the cells occupied by A for the presence of a B object (or vice-versa). To avoid near misses that can occur in such computations, objects are also installed in the 8-neighbors of each cell. That is, two objects can be very close together but happen to be in two distinct but adjacent cells. They should be considered near in such a case, and the strategy we have used is to look in the eight nearest neighbor cells. This, in effect, essentially "coarsens" the grid, but that can be overcome simply by using a finer grid (deeper pyramid).



Important relations such as horizontally-aligned can be computed immediately from the projection arrays. Though it is obvious how to compute predicates, such as, are A and B near one another?, it is often more important to generate sets of objects that satisfy a given spatial relation. This is also easily accomplished using SPAS -- to find all objects near A we form the union of all other occupants of cells occupied by A.

To accommodate spatial predicates that operate at various levels of resolution, SPAS is actually a pyramid of arrays of size 2ix2i, i=0,6 (the last value, 6, is a settable parameter) and the objects are installed in all levels of the pyramid. Generated sets are typically used to restrict the context that is passed to a constituent rule.

Multiple copies of higher-level objects may be constructed during a parse, but only one copy of a given constituent with identical leaf nodes is installed in the SPAS and referred to in the solution. This retains object integrity for all constituents. In the future, it may be possible to exploit this to even avoid rediscovering constituents, much as natural language chart parsers do. But diagram parsing cannot rely on the simple linear position index that natural language chart parsers do, so this will be a challenging efficiency issue. (We have experimented with memoization, but that also needs further work.)

One of the benefits of the spatial index is that once objects are installed, geometric computations can be done based on cells, ignoring the objects' geometric structure, so that it is as easy to find objects near a Bezier curve as it is to find objects near a straight line or a text item. Higher-level objects are installed in SPAS as the parsing process proceeds and each higher-level object occupies the cells that are the union of the cells occupied by the primitives descendants.

In the figure on the previous page, we show how the diagram viewer can display the cells occupied by an arbitrary object, in this case the higher-level object DATA-LINE.

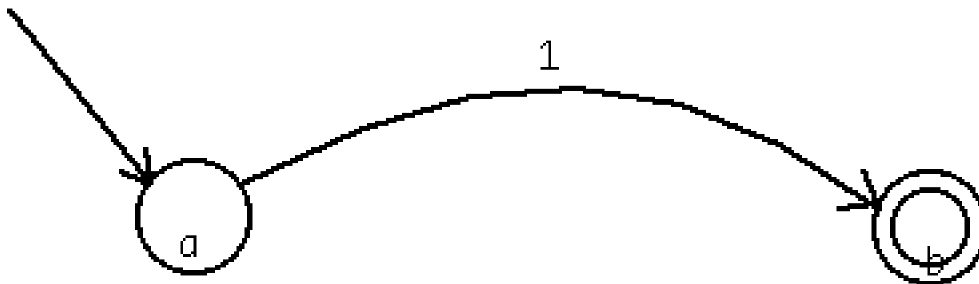
When lines or Bezier curves are installed, their endpoints are also installed as distinct but related objects. In the figure above it is clear that a connected sequence of data lines has been identified by the parser. This is done by using the spatial index to look in the neighborhood of a line end-point to see if the (starting) end-point of another line is there. In this way, a chain of connected lines or curves can be built up.

Finite-state automata diagrams -- using sharing for context

Here we exhibit some of the aspects of parsing the simple finite-state diagram below. The diagram arrows are simply drawn as a line or curve and two straight lines near the point, forming the arrowhead. The exact positioning, angles, and lengths of the arrowheads is not crucial to the analysis. The positioning of the three labels is not critical either. The full finite-state automata grammar is available for perusing. It may help clarify some of the details of the parsing whose results are illustrated below. Installation of the diagram in the pyramid required about 700 msec and parsing required about 250 msec. (This is relatively slow, given the simplicity of the diagram, but this is because the installation and parsing of circles is not well optimized.)

The grammar for finite-state automata uses existentially quantified variables and negation in the discovery of the start states. They must have an incoming arrow which itself has no other state touching its tail. This is specified in the rule below:

```
( Init-state -> A-state  Arrow
  (A-state)
  (Arrow  (touch A-state '?)
   :constraints  (touch (reach-pt Arrow) A-state)
                 (null (some* [a-state] :in
                  (touch '? (leave-pt Arrow))))))
```

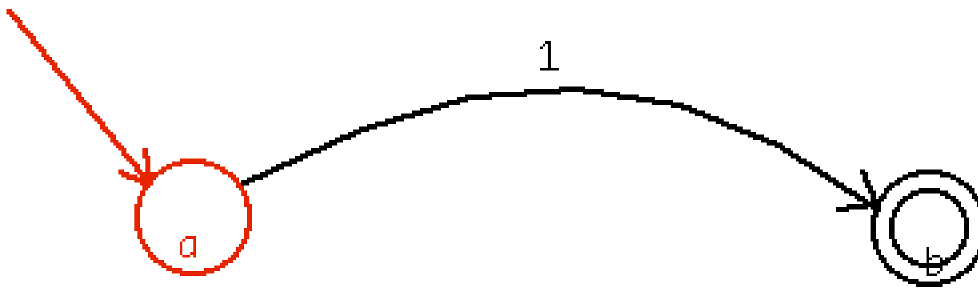


The top-level production has three constituents:

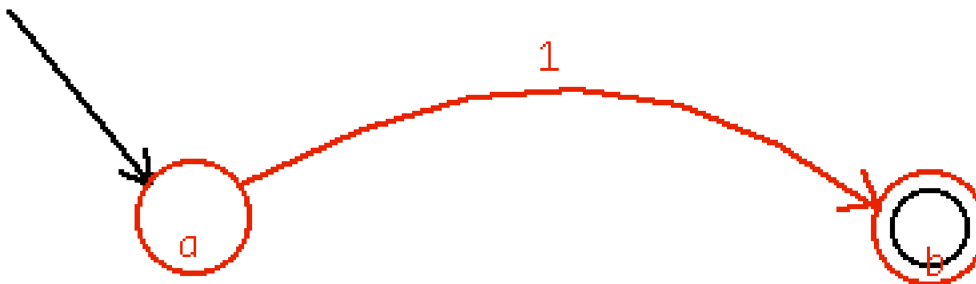
```
FA -> Init-state Transitions Final-states
```

Perhaps the most interesting thing about the grammar is that the three constituents share various elements. They do this so they can identify structures correctly by their relations to others, by their geometrical context. This is most easily seen in the three images below, which show the highlighted objects in the three top-level constituents.

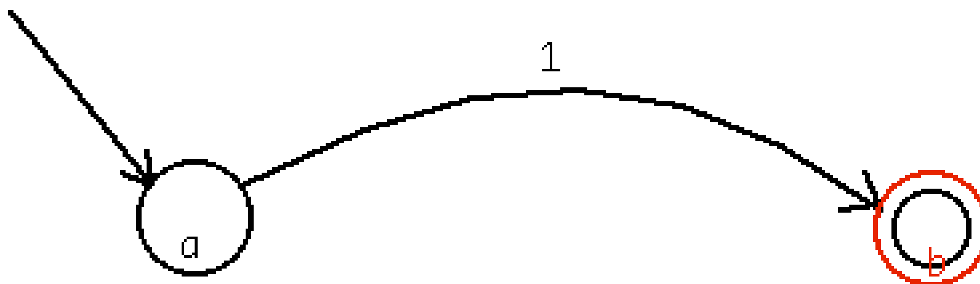
The figure below shows the highlighted elements of the Init-state. Both the state-circle and the incoming arrow without any tail object are used to determine that "a" is the initial state.



The figure below, shows highlighted elements of the Transitions (there is only one). Note that the circle in the initial state show in Fig. 1 is used again (shared) in defining the transition. Each state circle in a large diagram is a participant in all the transition objects that leave or enter that state. Because each entity such as a state is treated as a unique object, and installed in the pyramid, they are not recreated each time they are used in a distinct constituent parse.

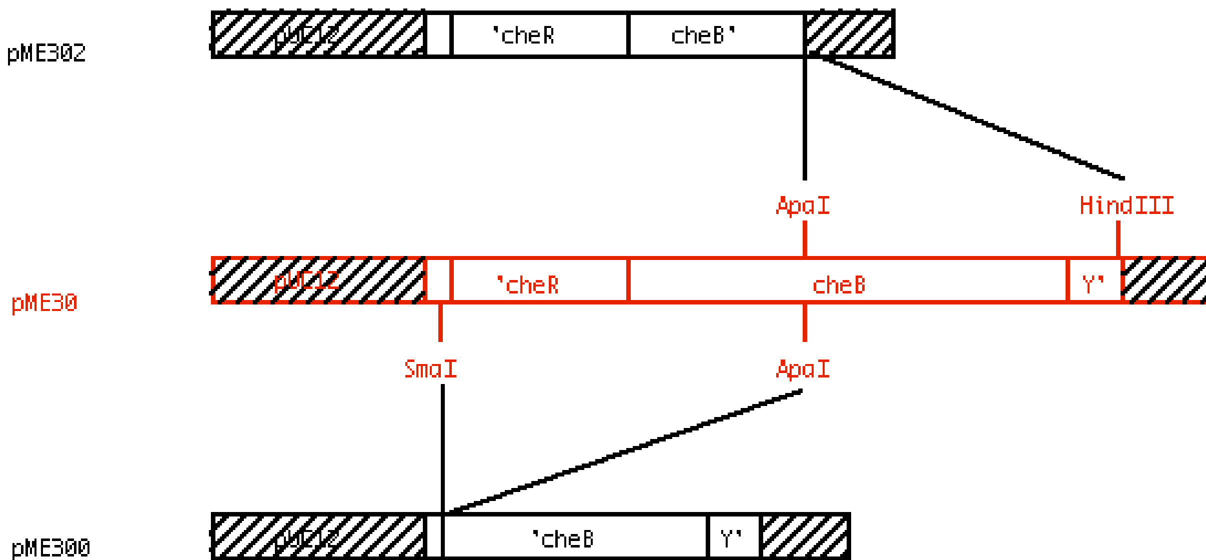


The figure below, shows highlighted elements of the Final-states (there is only one). A final state is distinguished by having one circle contained in another. Here the end-state of the transition highlighted in Fig. 2 is reused in the final state.



Gene Diagrams

Here we exhibit the results of parsing a gene diagram. The details are similar to the previous examples. The full gene grammar is available for perusing. Here is a gene diagram with one of the three genes found highlighted:



The primary components of a gene are the segments and an (optional) backbone line. The highlighted gene has three segments. Two are crosshatched and the third, middle segment, has further subdivisions. The grammar used had no provision for the crosshatching or the long vertical or diagonal connecting lines showing the correspondence between sites in the three gene variants in the figure.

Appendix A - Complete copy of a grammar for x,y data graphs

Note that the grammar is followed by definitions of some Lisp functions called during parsing, especially to deal with the details of arrow-points (arrowheads).

```
;;;-*- Mode: Lisp; Package: DUS -*-

;;; -----------------------------
;;;
;;; Nikos Nikolakis
;;;
;;; created:    Oct. 1994
;;; last update: 2/28/95
;;; -----------------------------

#|

   An effort to parse a diagram through a declarative approach.

   This grammar works fine for diagrams with non-overlapping axis.
   The grammar has been extended to handle special cases like
   annotations, key-specifications and other objects inside a diagram.

|#


(defvar *tiny* 35) ;  it specifies the coincide predicate (old value 20)

(setf

 *grammar*    ;; grammar object

 (defgrammar

;;; ***************** < Image > *****************

   ( Image -> Set ( Diagram ) );



;;; ***************** < Diagram > *****************

   ( Diagram -> Axis  X-Axis  Y-Axis  Data
      (Axis)
      (X-Axis   ($ :axis Axis))
      (Y-Axis   ($ :axis Axis))
      (Data
       ($ (difference*  (contain Axis '?)
                        (union* X-Axis Y-Axis))
          :x-ln (ln (X-Line Axis))
          :y-ln (ln (Y-Line Axis))
          :axis Axis)) );
```

```
;;; ****************** < Axis > ******************
   ( Axis -> X-Line   Y-Line
      (X-Line)
      (Y-Line    (touch (left-endpoint X-Line) '?)
       :constraints
       (coincide (left-endpoint X-Line)  (bottom-endpoint Y-Line))) );
   ( X-Line -> Line
      (:additional-slots   (left-endpoint . (left-endpoint (Line self)))
                           (ln            . (a-length (Line self))))
      (:constraints
       (horizp Line)  (long Line)) );
   ( Y-Line -> Line
      (:additional-slots   (bottom-endpoint . (bottom-endpoint (Line self)))
                           (ln              . (a-length (Line self))))
      (:constraints
       (vertp Line)   (long Line)) );



;;; ******************  < X-AXIS >  ******************

    ( X-Axis -> X-Axis-Line   X-Ticks   X-Labels  X-Annotation   X-Text
       (:optional X-Annotation X-Text)
       (X-Axis-Line  (X-Line (get-val axis)))
       (X-Ticks  ($ :x-line X-Axis-Line)  :constraints (>= (size X-Ticks) 3))
       (X-Labels     (below '? X-Axis-Line :strip t))
       (X-Annotation (difference* (near X-Axis-Line 700)
         (union* X-Ticks X-Labels))) ; label-size
       (X-Text       (near&below '? X-Labels (* 2 (height X-Labels)))
                                ) )

    ( X-Ticks ->  X-or < (Own-X-Ticks     (touch  '? (get-val X-Line)))
                 (Remote-X-Ticks  (below  '? (get-val X-Line) :strip t)) > );
    ( Own-X-Ticks ->  Set ( Line )
       (:element-constraints
        (vertp Line) (short Line)  (not (polylinep Line))
        (< (distance (endpoints Line) (Line (get-val X-Line)))
           (max (* (a-length Line) .25) *tiny*)))
       (:constraint horiz-aligned-gen)
       (:largest t) );
    ( Remote-X-Ticks ->  Set ( Line )
       (:element-constraints
        (vertp Line) (short Line)  (not (polylinep Line)))
       (:constraint horiz-aligned-gen)
       (:largest t) );

    ( X-Labels -> Set ( Text )
      (:element-constraints  (horizp Text)  (numeric-textp Text) )
      (:constraint horiz-aligned-gen)
      (:largest t) );

    ( X-Text -> Text
       (Text
        :constraints (horizp Text)  (not (numeric-textp Text))
```

```
        :select (max (text-length Text))) );

    ;;; The arrow points to the X-line (=> we don't need the pointed-gos
    ( X-Annotation ->  Arrow Text-set
       (:optional Text)
       (:non-sharable  Arrow  Text-set)
       (Arrow)
       (Text-set   (near  (leave-pt Arrow) (* 3.5 (ln Arrow)))) );

;;; ***************** < Y-AXIS > *****************

    ( Y-Axis -> Y-Axis-Line  Y-Ticks  Y-Labels  Y-Text
      (:optional Y-Ticks Y-Labels Y-text)
      (Y-Axis-Line       (Y-Line (get-val axis)))
      (Y-Ticks           ($ :y-line Y-Axis-Line))
      (Y-Labels          (left '? Y-Axis-Line :entirely nil :strip t))
      (Y-Text            ($ (left '? (or Y-Labels  Y-Axis-Line))
                            :dist (width Y-Labels))
       :select (min (distance (center Y-Axis-Line) (center Y-Text)))) );


    ( Y-Ticks ->  X-or < (Own-Y-Ticks     (touch '? (get-val Y-Line)))
                         (Remote-Y-Ticks  (left '? (get-val Y-Line) :strip t)
                           :filter   (some* [Y-Labels] :in (touch '? Remote-
Y-Ticks) :gen t)) > );


    ( Own-Y-Ticks ->  Set ( Line )
       (:element-constraints
        (horizp Line)  (short Line)
        (not (polylinep Line))
        (< (distance (endpoints Line) (Line (get-val Y-Line)))
           (max (* (a-length Line) .25) *tiny*)))
       (:constraint vert-aligned-gen)
       (:largest t) )


    ( Remote-Y-Ticks ->  Set ( Line )
       (:element-constraints
        (horizp Line) (short Line)  (not (polylinep Line)))
       (:constraint vert-aligned-gen) );


    ( Y-Labels -> Set ( Text )
      (:element-constraints (horizp Text) (numeric-textp Text) )
      (:constraint vert-aligned-gen)
      (:largest t) );


   ( Y-Text -> Set ( Text )
     (:element-constraints (vertp Text))
     (:constraint (close-gen :how-near (get-val dist))) );
```

```
;;; ****************** < DATA > ******************

    ( Data -> Data-lines  Data-points  Annotations Key-specifications
           (:optional Data-lines Data-points Annotations Key-specifications)
           (Key-specifications)
           (Data-lines)
           (Annotations  (difference* context Data-lines))
           (Data-points  (difference* context  Data-lines)) );


;;; ****************** < DATA-LINES > ******************

    ( Data-lines  -> set ( Data-line )
      (:element-constraints
       (> (a-length data-line) (* .25 (get-val x-ln)))));


    ( Data-line  -> set ( Line )
       (:element-constraints
        (not (polylinep Line))
        (not (or (and (horizp line) (> (a-length line)  (* .7 (get-val x-
ln)))
                      (< (distance (left-endpoint line)
                                   (Line (Y-Line (get-val axis)))) *tiny*))
                 (and (vertp  line) (> (a-length line)  (* .7 (get-val y-
ln)))))))
       (:constraint connected) ) ;


    ( Data-line  -> set ( Curve ) (:constraint connected)) ;


;;; ****************** < Data-points > ******************

    ( Data-Points -> set ( Data-Cluster ) ); ={ circle-dp, traingle-dp,
rectangle-dp }


    ; An example of a data-cluster is the set of all circle-data points
    ( Data-Cluster -> set ( Data-point ) (:constraint same-type) ) ;


    ( Data-point -> circle
      (:constraints  (whitep circle)) ); a white circle


    ( Data-point -> circle
      (:constraints  (blackp circle)) ); a black circle


    ( Data-point -> polygon  ; a rectangle
      (:constraints  (rectangle? polygon) (small polygon)) ) ;
```

```
    ( Data-point -> line_1  line_2  line_3     ; a triangle
       (line_1)
       (line_2       (touch '? (right-endpoint line_1))
         :constraints
         (different line_1 line_2)
         (coincide (right-endpoint line_1) (left-endpoint line_2)))
       (line_3       (touch '? (right-endpoint line_2))
        :constraints
        (different line_2 line_3) (different line_1 line_3)
        (coincide  (right-endpoint line_2) (right-endpoint line_3))
        (coincide  (left-endpoint  line_1) (left-endpoint  line_3))) );


    ( Data-point -> line_1 line_2 line_3 line_4 ; a diamond
        (line_1
          :constraints (not (horizp line_1))  (not (vertp line_1)))
        (line_2   (touch '? (right-endpoint line_1))
          :constraints
          (< (distance (right-endpoint line_1) (left-endpoint line_2)) *ti-
ny*)
          (not (horizp line_2))
          (not (vertp line_2)))
        (line_3  (touch '? (right-endpoint (line_2 obj)))
          :constraints
          (< (distance (right-endpoint line_2) (right-endpoint line_3)) *ti-
ny*)
          (different line_3  line_2))
        (line_4   (touch '? (left-endpoint line_3))
          :constraints
          (< (distance (left-endpoint line_3) (right-endpoint line_4)) *ti-
ny*)
          (< (distance (left-endpoint line_1) (left-endpoint line_4)) *tiny*)
          (parallelp line_1 line_3)
          (above (right-endpoint line_1) (right-endpoint line_4))) );

;;; ***************** < Annotations > *****************

   (Annotations -> set ( Annotation ) );


   ;;; When a new set-context is specified (in a normal rule), it overwrires
the previous one.
   ;;; //  extend relation for set rules.
   ;;; We could have done: (gen (intersect old-context new-fun))).

   (Annotation ->  Text-set  Arrow  Pointed
     (:optional Text Pointed)
     (Arrow)
     (Text-set   (intersect* (near  (leave-pt Arrow) (* 2 (ln Arrow)))
                             context)
                ;:select (min  (distance (center Text)  (line (arrow-back
Arrow))))
                )
```

```
    (Pointed  (near (pt (arrow-head arrow)) (* 2 (a-length (Line (Arrow-back
Arrow)))))) );


   ( Text-set -> set ( Text )
     (:element-constraints
      (horizp Text) (not (numeric-textp text))) );


   ( Pointed -> set ( Object ) );


   ( Object -> Data-point );


   ( Object -> Text );


;;; ****************** < Arrow > ******************

   ( Arrow-back -> Line
     (:additional-slots
      (left-endpoint  . (left-endpoint  (Line  self)))
      (right-endpoint . (right-endpoint (Line  self)))
      (ln             . (a-length (Line self))))
     (:constraints  (not (polylinep line))) ); <---


   ( Arrow-head -> Line_1  Line_2  ; works OK
     (:additional-slots
      (pt . (get-arrow-head-pts self 'common-pt))
      (p1 . (get-arrow-head-pts self 'p1))
      (p2 . (get-arrow-head-pts self 'p2)))
     (Line_1
      :constraints (short Line_1) (not (polylinep Line_1)))
     (Line_2  (touch Line_1 '?)
      :constraints
      (left  (center Line_1) (center Line_2)) ; impose ordering
      (short Line_2)
      (not (polylinep Line_2))
      (same-length  Line_1  Line_2 :ratio 1.2)
      (< (distance (endpoints Line_1) (endpoints Line_2)) 40)) );


   ( Arrow -> Arrow-back Arrow-head ; works OK
           (:non-sharable  Arrow-head)
     (:additional-slots
      (leave-pt . (if (> (distance (pt (Arrow-head self))
                                (left-endpoint (Arrow-back self)))
                         (distance (pt (Arrow-head self))
                                (right-endpoint (Arrow-back self))))
                    (left-endpoint (Arrow-back self))
                       (right-endpoint (Arrow-back self))))
      (reach-pt . (pt (arrow-head self)))
```

```
      (ln              . (ln (Arrow-back self))))
     (arrow-head)
     (arrow-back  (intersect* (near (pt arrow-head) (height Arrow-head))
         context) ;;; <-- put line_1
      :constraints
      (in-angle (pt Arrow-head) (p1 Arrow-head) (p2 Arrow-head)
                   (left-endpoint Arrow-back))
      (in-angle (pt Arrow-head) (p1 Arrow-head) (p2 Arrow-head)
                   (right-endpoint Arrow-back))
      (< (distance (pt Arrow-head) (endpoints (line Arrow-back)))
         (* 1.5 (a-length (line_1 arrow-head))))
      (same-angle (angle (left-endpoint Arrow-back) (pt Arrow-head))
                  (angle (right-endpoint Arrow-back) (pt Arrow-head)))
      :select
      (max (a-length (line Arrow-back)))) );

;;; ************** < Key-specifications > ******************

     ( Key-specification -> Expl Data-Points
            (:non-sharable Data-Points)
            (Expl  :constraints (>= (size Expl) 2))
            (Data-Points  (near Expl 300)) );

     ( Expl -> Set ( Text )           ;  Left-Aligned text
        (:element-constraints
          (horizp Text) (not (numeric-textp Text)) (> (text-length Text) 3))
            (:constraint (vert-aligned-gen :left t)) );

     ( Expl -> Set ( Text )            ;  Right-Aligned Text
        (:element-constraints
          (horizp Text) (not (numeric-textp Text)) (> (text-length Text) 3))
           (:constraint (vert-aligned-gen :right t)) );

        ( Key-specifications -> Set ( Key-specification ) );
   ))

;;; -------------------------------------------------------
;;;   Terminal generators for non-primitive terminals
;;; -------------------------------------------------------
;;;
(defun gen-x-axis-line (context)
  (list context))
;;;
(defun gen-y-axis-line (context)
  (list context))
;;;
(defun coincide (x y &key (dist *tiny*))
  (< (distance x y) dist))

;;; -------------------------------------------------------
;;;         Non-supported relations
;;; -------------------------------------------------------

(defmethod a-length ((data-line data-line))
```

```
  (data-line-length data-line))
(defmethod data-line-length ((data-line data-line))
  (apply #'+
         (mapcar
          #'(lambda (x) (distance (left-endpoint x) (right-endpoint x)))
          (value data-line))))
(defun distance-pt-from-data-line (x dl)
  (distance-pt-from-lines x (value dl)))
;;;
(defmethod get-arrow-head-pts ((ob arrow-head) type)

  (let* ((line1-end-pts (make-array '(2)
                                    :initial-contents (a-line-terminators
(line_1 ob))))
         (line2-end-pts (make-array '(2)
                                    :initial-contents (a-line-terminators
(line_2 ob))))
         (distance-array (make-array '(2 2) :initial-element nil)))

    (dotimes (i 2)
    (dotimes (j 2)
      (setf (aref distance-array i j)
            (distance (aref line1-end-pts i) (aref line2-end-pts j)))))

    (let (common-pt f-p1 f-p2 min id-1 id-2);
      ;; common-pt corresponds to the "corner" point of the arrow head

      (setf min (aref distance-array 0 0))

      (dotimes (i 2)
      (dotimes (j 2)
        (when (<= (aref distance-array i j) min)
          (setf min (aref distance-array i j))
          (setf id-1 i)
          (setf id-2 j))))

    (setf common-pt (aref line1-end-pts id-1))
    (setf f-p1 (aref line1-end-pts (- 1 id-1)))
    (setf f-p2 (aref line2-end-pts (- 1 id-2)))

    (cond ((equal type 'all) (list common-pt f-p1 f-p2))
          ((equal type 'common-pt) common-pt)
          ((equal type 'p1) f-p1)
          (t f-p2)))))
```

Appendix B - Full Finite-State Automata Grammar

This is the grammar used to parse the simple example shown, and other more complex, many-state diagrams, not shown. This grammar uses specialized Lisp functions, appended the end of this grammar, to describe the detailed geometrical arrangement of the component of arrowheads.

```
(setf *grammar*    ; a grammar object

  (defgrammar

   ;;; **********  < Finite Automato >  *************
   ( FA -> Init-state  Transitions  Final-states
        (Transitions)  (Final-states)  (Init-state) ) ;

   ;;; **********  < Transitions > *************
   ( Transition -> A-state_1  Labeled-arrow  A-state_2
      (Labeled-arrow)
      (A-state_1  (touch (leave-pt (arrow Labeled-arrow)) '?))
      (A-state_2  (touch (reach-pt (arrow Labeled-arrow)) '?)) ) ;

   ( Transitions -> set ( Transition ) ) ;

   ;;; **********  < Labeled-arrow > *************
   ( Label -> Text
     (Text :constraints  (numeric-textp Text)) ) ;

   ( Labeled-arrow -> Arrow Label
     (Arrow)
     (Label  (touch  Arrow '?)
             :select (min (distance (center Label) (arrow-back Arrow)))) ) ;

 ;;; **********  < Arrow >  *************
   ( Arrow -> Arrow-back  Arrow-head
     (:additional-slots (leave-pt . (get-leave-pt self))
                        (reach-pt . (pt (arrow-head self))))
     (Arrow-head)
     (Arrow-back   (extends (touch (pt arrow-head) '?))
                   :select (min (distance (list (left-endpoint arrow-back)
                                      (right-endpoint arrow-back))
                               (pt arrow-head)))) ) ;

   ;;; **********  < Arrow-head >  *************
   ( Arrow-head -> Short-line_1  Short-line_2
     (:additional-slots  (pt . (get-arrow-head-pts self 'common-pt)))
     (Short-line_1)
     (Short-line_2  (touch Short-line_1 '?)
             :constraints
             (different Short-line_1  Short-line_2)
             (left  (center Short-line_1)  (center Short-line_2) :strictly t)
             (same-length (line Short-line_1) (line Short-line_2) :ratio 1.7)
```

```
             (< (distance (endpts short-line_1) (endpts short-line_2))
                (/ (a-length (line Short-line_1)) 5.0))) ) ;  ~ 40

    ;;; **********  < Arrow-back > *************
   ( Arrow-back -> Line
     (:additional-slots  (left-endpoint  . (left-endpoint  (Line  self)))
                         (right-endpoint . (right-endpoint (Line  self))))
     (:constraints  (long Line :ratio 22))) ;

   ( Arrow-back ->  Curve-back
     (:additional-slots  (left-endpoint .(left-endpoint  (Curve-back self)))
                         (right-endpoint .(right-endpoint (Curve-back self)))
     )) ;

   ( Curve-back -> set ( Curve ) (:constraint connected)) ;

   ;;; **********  < Short-line > *************
   ( Short-line -> Line
     (:additional-slots  (endpts . (endpoints (Line self)))
                         ;(length . (a-length (Line self)))
                         ) ;; add length
     (:constraints (short Line))) ;

   ;;; **********  < A-state > *************
   ( A-state -> Circle  Text
     (Circle :constraints (not (contained Circle (some* [circle]))))
     (Text   (contain Circle '?))) ;

   ;;; **********  < Final-states >  *************
   ( Final-state -> A-state
      (A-state
       :constraints (contain (circle  A-state) (some* [circle] :in (touch '?
A-state))))) ;

   ( Final-states -> set ( Final-state )) ;

 ;;; **********  Init-state  *************
 ( Init-state -> A-state  Arrow
   (A-state)
   (Arrow  (touch A-state '?)
    :constraints (touch (reach-pt Arrow) A-state)
                 (null (some* [a-state] :in (touch '? (leave-pt Arrow))))));
   ))

;;; -------------------------------------------
;;;         Non supported functions
;;; -------------------------------------------

(defmethod left-endpoint ((go curve-back))
  (apply #'left-point (curve-cluster-endpoints (value go))))

(defmethod right-endpoint ((go curve-back))
  (apply #'right-point (curve-cluster-endpoints (value go))))
```

```
(defmethod distance ((pt a-point) (go arrow-back<1>) &key (min-max 'min))  ;
<1> the first alternative in the grammar.
  (distance pt (line go)))

(defmethod distance ((pt a-point) (go arrow-back<2>) &key (min-max 'min))
  (pt-curve-cluster-dist pt (value (curve-back go))))

;;; ----------------------------------------
(defmethod get-arrow-head-pts ((ob arrow-head) type)

  (let* ((line1-end-pts (make-array '(2)
           :initial-contents (a-line-terminators (line (short-line_1 ob)))))
         (line2-end-pts (make-array '(2)
           :initial-contents (a-line-terminators (line (short-line_2 ob)))))
         (distance-array (make-array '(2 2) :initial-element nil)))

    (dotimes (i 2)
    (dotimes (j 2)
      (setf (aref distance-array i j)
            (distance (aref line1-end-pts i) (aref line2-end-pts j)))))

    (let (common-pt f-p1 f-p2 min id-1 id-2);
      ;; common-pt corresponds to the "corner" point of the arrow head

      (setf min (aref distance-array 0 0))

      (dotimes (i 2)
      (dotimes (j 2)
        (when (<= (aref distance-array i j) min)
          (setf min (aref distance-array i j))
          (setf id-1 i)
          (setf id-2 j))))

    (setf common-pt (aref line1-end-pts id-1))
    (setf f-p1 (aref line1-end-pts (- 1 id-1)))
    (setf f-p2 (aref line2-end-pts (- 1 id-2)))

    (cond ((equal type 'all) (list common-pt f-p1 f-p2))
          ((equal type 'common-pt) common-pt)
          ((equal type 'p1) f-p1)
          (t f-p2)))))

(defun get-head-pt (arrow-head)
  )

(defun get-leave-pt (arrow)
  (let ((pt (pt (arrow-head arrow))))

    (if (> (distance pt (left-endpoint  (arrow-back arrow)))
           (distance pt (right-endpoint (arrow-back arrow))))
    (left-endpoint (arrow-back arrow))
    (right-endpoint (arrow-back arrow)))
    ))
```

Appendix C - Full Gene Grammar

This is the grammar used to parse the three part gene shown earlier.

```
;;; -------------------------------------------------
;;;
;;; Created: 11/7/94
;;;
;;; Nikos Nikolakis
;;;
;;;  A general grammar for parsing of Gene diagrams
;;; -------------------------------------------------

(defvar *sline-ratio* 6)
(defvar *small* 100)
(defvar *large* 1050)

(setf *grammar*

 (defgrammar

   ( Gene-Diagram -> Set ( Gene ) );

   ;;; ******************** GENE ********************

   ( Gene -> Gene-body  Gene-Title  Tick-specifs
     (:optional Gene-Title Tick-specifs)
     (Gene-body)
     (Gene-Title  (difference* (intersect* (left '? Gene-body :entirely nil)
                                           (horiz-aligned-gen Gene-body
                                     :how-near (/ (height Gene-body) 1.5)))
                               (Backbone Gene-body)))
     (Tick-specifs  ($ (intersect* (touch '? Gene-body)
                            (above&below '? Gene-body :strip t))
                                :segments (Segments Gene-body))));

   ( Gene-Title -> Set ( Text ) );

   ( Gene-body -> Segments  Backbone
     (:optional Backbone)
     (Segments)
     (Backbone   (touch Segments '? :every t)
       :constraints
       (left (left-endpoint (Line Backbone)) Segments)) );

;;; ***************** Tick-specifs *****************

   ( Tick-specifs -> set ( Tick-specif ) );

   ( Tick-specif -> Line  Close-or-Remote-Label
      (Line :constraints (vertp Line) (short Line) (not (polylinep Line)))
```

```
      (Close-or-Remote-Label ($ :line Line)) );

   ( Close-or-Remote-Label -> X-or < (Label (touch (get-val Line) '?))
                                     (Label (vert-strip (get-val Line))) > );

   ( Label -> Text
           (Text
            :constraints
            (not (member Text (solution->list (get-val Segments))))
                   ; ignore segment-title
            (or (and (below (center (get-val Line))
                   (center (get-val Segments))) (below (center Text)
                        (center (get-val Segments))))
                (and (above (center (get-val Line))
                   (center (get-val Segments))) (above (center Text)
                        (center (get-val Segments)))))
            :select
            (min (min (abs (- (a-point-x (center (get-val Line)))
                              (a-point-x (center Text))))
                      (abs (- (a-point-x (center (get-val Line)))
                              (a-point-x (ur-point Text))))
                      (abs (- (a-point-x (center (get-val Line)))
                              (a-point-x (ll-point Text))))))));

;;; ******************** BACKBONE ********************

( Backbone -> Line  Left-label  Right-label
   (:optional Left-label  Right-label)
   (Line  :constraints (horizp Line)  (long Line)  (not (polylinep Line)))
   (Left-label   (touch '? (left-endpoint Line))
                 :select (min (distance (center Left-label)
                                (left-endpoint Line))))
   (Right-label (touch '? (right-endpoint Line))
                :select (min (distance (center Right-label)
                                       (right-endpoint Line)))));
( Left-label -> Text );

( Right-label -> Text );

;;; ******************** SEGMENTS ********************
   ( Segments -> Set ( Segment )
       (:constraint  horiz-aligned-gen) ); <--

   ( Segment -> Body  Divisions
      (:optional Divisions)
      (Body :constraints  (< (height Body) *large*))
      (Divisions  ($ (contain Body '?)  :body Body  :width (width Body))) );

;;; ******************** BODY ********************
   ( Body -> Polygon   ; rectangle body
      (:constraints  (rectangle? Polygon)  (< (height Polygon) *large*)) );

   ( Body -> Line_1  Line_2  ; body made from two horizontal lines
```

```
    (Line_1
       :constraints
        (horizp Line_1)  (long Line_1 :ratio 7)  (not (polylinep Line_1)))
     (Line_2  (near Line_1 (/ (a-length Line_1) 4.0))
        :constraints
        (horizp Line_2)  (long Line_2 :ratio 7)  (same-length Line_1 Line_2)
        (not (polylinep Line_2)) (below (center Line_2) (center Line_1))));

;;; ******************* DIVISIONS *******************
  ( Divisions -> Set ( Division ) );

  ( Division -> Division-Marks  Division-Title
     (:optional Division-Title)
     (Division-Marks :constraints  (> (width Division-marks) *small*))
     (Division-Title
          ($ (near (get-val Body) (get-val width))  :box Division-Marks)) );

  ( Division-Marks -> Pair ( Line )
      (:element-constraints (vertp Line)  (contained  Line (get-val body)))
      (:constraint  neighbor-pairs :direction 'x) );

;;; ******************* DIVISION-TITLE *******************
   ( Division-Title -> Set ( Text )
      (:element-constraints  (contained (center Text) (get-val box))));
   ))
```

Appendix D - Relation of this to ideas/systems in FRVDR98

This page contains discussions of some of the points of contact between this work and the work in the FRVDR98 Fall Symposium, as recorded in the Proceedings. See the Contents list for the symposium at the end of this page. My comments here will be reasonably self-contained, but they will be more meaningful to someone who has a copy of the proceedings in hand.

Free Rides and SPAS -- "Free rides" is a concept that is mentioned by various papers, e.g., (Gurr), and refers to Shimojima's 1996 paper. It means that when viewing a diagram, certain inferences are available directly by inspection of the diagram, and follow directly from the spatial layout of the diagram. Examples include Euler and Venn diagrams. Our SPAS spatially associative data structure can be thought of from this viewpoint. If you want to know if a certain object passes near a certain small region in space, you simply look in the corresponding SPAS cell to see if it contains a reference to the object. That is, SPAS directly represents the near relation -- it is a type of spatial cache. (An alternate representation could store all objects near to some object O in a near list attached to O -- another type of cache. The near lists would not necessarily have size N2 since, for example, a set of well-spaced points would have zero entries in all their near lists.) Similarly, if we want to know if a set of points is horizontally aligned, we can see if they all appear in some cell in the y projection array of SPAS.

Generalized equivalence relations (GERs) -- We have stressed in the past that human perception is one of the sources of free rides, (Futrelle, 1990). For example, a collection of aligned items is a "pop-out" phenomena, as evidenced by the existence of illusory contours (also called subjective contours), (Petry, 1987). These occur when for example, a set of small objects arranged in space to form a broken circumference of a circle, which lead to the illusion of an actual circle whose interior is lighter than the surrounding plane. A related phenomenon is the pop-out phenomena in which psychophysical experiments have determined that people can detect the presence of an "O" in a collection of "X" distractors in the visual field in a time that is independent of the number of distractors (Treisman, 1985, 1990). Thus, the extraction of such information is a "free-ride" in that it is done as a pre-attentive parallel operation by the visual system. In general, we refer to uniform sets of objects that are collected together as an equivalence set or an extension of the equivalence relation such as near or equally spaced, as obeying a Generalized Equivalence Relation. Another example is a collection of data points in a region of a graph all of the same size and shape, e.g., triangles. Our algorithm normally chooses the maximal collection of the possible sets of the elements so related. This choice can also be argued on entropy grounds and more generally by minimal-length encoding theory or MLE (Li and Vitanyi, 1993).

Local extent -- (Foo) discusses the importance of local extent. Since diagrams are finite objects, direct reasoning about them must have local extent. Again, we have developed SPAS to aid such reasoning.

Perception and understanding -- (Hayes and Laforte) have an interesting and provocative paper that discusses the relation between perception and understanding using variations on the diagrammatic proof of Pythagoras' theorem. One reason this is important is that parsers such as ours are faced with the problem of geometric inference. Consider the following innocent example, analogous to a class of problems met in diagram parsing -- what are the letter-number pairings in the following two sequences: b 40 k 12 p 32 versus 36 b 40 k 12 p? The sequences agree precisely in the central subsequence b 40 k 12 p, but the first sequence clearly groups as (b 40) (k 12) (p 32) and the second, (36 b) (40 k) (12 p). This is a potential, but in fact, resolvable, ambiguity, and some reasoning is required to establish the grouping. The grammar-based approaches to parsing are not well-suited for such tasks.

Scaffolding -- In (Allwein) it is pointed out that the Greek approach to geometry often involved auxillary constructions or "scaffolding". In our grammars, because of the controlled sequence of top-down expansion of productions, we can introduce "scaffolding" and then use it to support the discovery of the constituents of interest. This is done, for example, in the Diagram rule:

```
Diagram -> Axis X-Axis Y-Axis Data
```

where Axis is a non-terminal that is required to be made up of a long horizontal line and a long vertical line which touch at the lower left (details here). Once Axis is determined in this way, the remaining constituents can refer to its components in their productions, e.g., extracting the long horizontal line from Axis for use in the X-Axis production.

Relation of Chok and Marriott's work to ours -- Perhaps the most obvious point of contact between our work and work described in the symposium, is Chok and Marriott's paper on the Penguins project. They build parsers for diagrams that can operate in batch mode or incrementally in a pen-based drawing application. They also have a solution to the layout problem, error detection, and more. This is excellent work, which in some ways makes our work seem modest by comparison. But our work has very different origins and goals, which explains at least some of the differences. Our projects have focused on understanding diagrams from the research literature, so our approach to parsing and the examples we parse have been chosen accordingly. Nevertheless, the grammatical formalism we use is very close to Chok and Marriott's (due in part to the fact that we studied Marriott's work, among others, early in the development of our system some time ago). Our system uses both synthesized and inherited attributes. The major inherited attribute (passed down the tree as parsing proceeds) is the context, which is the set of primitive objects that can be used by a rule in searching for a solution, an idea not in their work. The synthesized attributes, passed up the tree, are not restricted to being chosen from attributes of the primitives. For example, we could compute and pass up the smallest distance between any two of the points in a set of n points found in solving a particular production, or the spacing of a set of equally spaced tick marks. As they do, we implement negative constraints -- the example we mentioned earlier was (not (polylinep Line)) in which we want to ignore any lines that are a side of a polygon. The relation between their grammar for finite-state automata and our gram-

mar is very close. But our rule for defining a final state is notably simpler than theirs because we use the contain constraint, that one circle be contained within another, rather than object-type-specific computations involving the centers and radii of the two circles.

Our parser is also notably slower than theirs, even on a fast machine. But they have focused on interactive applications in which speed is of the essence, and have used Borning's highly refined constraint solver, itself designed for on-line operation. We have been more concerned with expressiveness than speed, and undoubtedly could speed up our system by an order-of-magnitude, were that required. But the literature of science that is our focus is finite and diagrams in the hard sciences appear in the published literature at a rate of a few million per year. An upper bound would be 10 million distinct diagrams published per year. Since there are 32 million seconds per year, all diagrams in the published literature could be parsed, as fast as they are produced, by a parser that could parse a diagram in three seconds, on average. The more important questions have to do with the goals of parsing diagrams. For us, it is the building of knowledge bases that will give users access the huge collections of diagrammatic information that is contained in the research literature of science and engineering.

One of the operations Chok and Marriott's system can do is related to unrestricted productions which can have more than one left-hand-side symbol. In their Figure 2, they create four lines from two intersecting lines. Interestingly, this brings up a major paradox that they do not discuss, which we might call the occlusion paradox. Given a collection of primitive objects in a representation of an image, e.g., a file listing the primitives and their attributes, the appearance of the diagram may differ notably from the representation. In the two intersecting lines case, it is impossible for a human observer to know whether there are two intersecting lines, or four lines with some coincident endpoints, or indeed, two L-shaped broken lines with a common corner. Sometimes, when people draw diagrams using an interactive drawing application, they deliberately use occlusion to produce a particular visual effect. This is often done because of the limitation of the drawing application. The extreme example of the paradox is the blank sheet. In this example, an arbitrarily complex drawing is created. Then a large white opaque, borderless rectangle (or circle or whatever) is drawn with dimensions large enough and position chosen to totally cover all elements in the original drawing. The parser should report an empty image, but no parser that I know of, including ours, would return such an interpretation.

I will have more to say about Chok and Marriott's important paper when I act as the discussant for it at the Symposium. The reader is also referred to the recent extensive survey on visual language specification and recognition (Marriott, Meyer, and Wittenburg, 1998).

Bibliography

Allwein, G., Marriott, K., & Meyer, B. (1998). Formalizing Reasoning with Visual & Diagrammatic Representations. Rept. No FS-98-04. AAAI Press.

Futrelle, R. P. (1990). Strategies for Diagram Understanding: Object/Spatial Data Structures, Animate Vision, and Generalized Equivalence. In 10th ICPR, (pp. 403-408): IEEE Press.

Li, M., & Vitanyi, P. (1993). An Introduction to Kolmogorov Complexity and Its Applications. New York: Springer-Verlag.

Marriott, K., Meyer, B., & Wittenburg, K. (1998). A Survey of Visual Language Specification and Recognition. In K. Marriott & B. Meyer (Eds.), Visual Language Theory (pp. 5-85): Springer Verlag.

Petry, S., & Meyer, G. E. (Ed.). (1987). The Perception of Illusory Contours: Springer-Verlag.

Shimojima, A. (1996). Operational Constraints in Diagrammatic Reasoning. In G. Allwein & J. Barwise (Eds.), Logical Reasoning with Diagrams (pp. 27-48). New York: Oxford.

Treisman, A. (1985). Preattentive Processing in Vision. Computer Vision, Graphics, and Image Processing, 31, 156-177.

Treisman, A., & Sato, S. (1990). Conjunction Search Revisited. Journal of Experimental Psychology: Human Perception & Performance, 16(3), 459-478.

Excerpt from the online description of the technical report of the symposium:
Formalizing Reasoning with Visual and Diagrammatic Representations
Papers from the 1998 Fall Symposium
Gerard Allwein, Kim Marriott and Bernd Meyer, Cochairs

October 23-25, Orlando, Florida

Technical Report FS-98-04
112 pp., $25.00
ISBN 1-57735-078-2

Visual Language Specification and Recognition
Kim Marriott 1

Theories of Visual and Diagrammatic Reasoning: Foundational Issues

Corin A. Gurr 3

Diagrammatic Reasoning and Color
Michael Anderson and Chris Armen 13

Verification of Diagrammatic Proofs
Mateja Jamnik, Alan Bundy, and Ian Green 23

Diagrammatic Reasoning
Gerard Allwein 31

Diagrammatic Reasoning: Analysis of an Example
Patrick J. Hayes and Geoffrey L. LaForte 33

Diagrammatic Reasoning about Actions Using Artificial Potential Fields
Marcello Frixione, Gianni Vercelli, and Renato Zaccaria 39

Local Extent in Diagrams
Norman Foo 51

A Logic-based Formalism for Reasoning about Visual Representations
Volker Haarslev 57

Generating User Interfaces for Pen-based Computers
Sitt Sen Chok and Kim Marriott 67

Hypergraph Representations of Diagrams in Diagram Editors
Mark Minas 79

Euclid++
Gerard Allwein 87
System Demonstrations

Diamond: Diagrammatic Reasoning System Demonstration
Mateja Jamnik, Alan Bundy, and Ian Green 95

The BITPICT Computation System
George W. Furnas 97

Demonstration of the Diagram Understanding System
Robert P. Futrelle 99

GenEd: A Generic Editor for Reasoning about Visual Notations
Volker Haarslev and Michael Wessel 101

VISCO: Querying GIS with Spatial Sketches

Volker Haarslev and Michael Wessel 103

Inter-Diagrammatic Reasoning
Michael Anderson 105

Interpretation of Visual Notations in the Recopla Editor Generator
Bernd Meyer and Hubert Zweckstetter 107