GXF: A Graph Exchange Format

Frank Ch. Eigler
fche@db.toronto.edu

January 13, 1993

Contents

1 Introduction **2**

2 GXF file syntax **2**
2.1 Introduction 2
2.2 Character set 2
2.3 Detailed syntax 2
2.4 Atoms 3
2.5 Whitespace 3
2.6 Comments 3

3 GXF file structure **4**
3.1 Introduction 4
3.2 GXF structures 4
3.3 Graphical co-ordinates 11

4 GXF samples **11**
4.1 A sample GXF file 11
4.2 Internet GXF archive 13

5 Hy+ requirements **13**
5.1 Introduction 13
5.2 Labels in databases 16
5.3 Labels in queries 16

1 Introduction

Graph Exchange Format ("GXF") is a specification for a portable external representation for directed graphs and "hygraphs" (defined in [4]). GXF allows graph processor programs to operate on data encoded in a common format which includes certain essential features of graphs. In addition, GXF lets a graph processor program to add any arbitrary information to a GXF file in a consistent, portable manner that is transparent to other programs which do not expect the extra information.

Hy+ is a tool for visualizing, editing, and querying hygraph-based representations of databases [1] [3]. **Hy+** can read/write graphs in GXF format from/to UNIX files, and uses extensions of GXF to store all layout and display-related information. GXF processor programs may use these **Hy+** extensions to manipulate the visual representation of a graph.

This document includes the definitions of GXF syntax, semantics, comments on processing requirements, and some samples. At the end, we give a short summary of what extra requirements **Hy+** imposes on GXF files for it to use them as databases or queries.

User-level familiarity with **Hy+** is useful in order to understand the use of visualization-related parameters stored in GXF files, but section 4 includes screendumps that demonstrate the use of some of these parameters by **Hy+** .

2 GXF file syntax

2.1 Introduction

GXF syntax is very similar to LISP syntax. A GXF file is structured into *lists* which contain multiple *atoms* and *lists*. The GXF format defines the interpretation of the contents of the lists.

2.2 Character set

Files in GXF format use a subset of the ASCII character codes: the printable characters and these of the standard separators: SPACE (32), TAB (9), and LF (10). To maximize portability, no other control characters are present, including the CR character (13). GXF does not support non-UNIX end-of-line conventions.

2.3 Detailed syntax

GXF files use the following lexical grammar, in standard notation:

GXF-file	:=	list {list}	
list	:=	`'('` keyword-atom {argument} `')'`	
keyword-atom	:=	*keyword*	(see section 2.4)
argument	:=	list	
	|	string-atom	
	|	number-atom	
string-atom	:=	*string*	(see section 2.4)
number-atom	:=	*number*	(see section 2.4)

where

[object]	==	zero or one instance of 'object'
{object}	==	zero or more instances of 'object'
`'('`	==	literal open-parenthesis (ASCII 40)
`')'`	==	literal close-parenthesis (ASCII 41)
atom	==	a 'atom' type literal

2.4 Atoms

The *keyword* atom is an alphanumeric string that starts with a letter and may contain letters, digits, and the underscore character. The keyword, in the context of its containing lists, identifies the data contained in the list. It is interpreted in a case-independent manner, eg.,

```
Wombat23
A_B_C_5A
WOMBAT23 ; (treated identically as the first item)
```

The *number* atom is a real number coded as a decimal floating-point constant, with no range or precision limits. It is written in the "standard" C/Fortran notation. Note the use of `E` or `e` as the exponent specifier. The unary `+` sign is not supported in either the mantissa or the exponent, eg.,

```
2.543
2E925 ; (a rather large number)
-3.500001
```

Processor programs should use numeric variables with the largest possible range and precision, and substitute those incoming *number* atoms which fall outside the program's usable range, by closest numbers within the usable range. If precision and accuracy is important, it is *suggested* that GXF files contain only numbers that can be well represented by standard C `float` or `int` variables.

The *string* atom is a string of ASCII characters enclosed in double-quotes (`"`) (ASCII 34). The string is uninterpreted except for the character sequences `\"` and `\\` represent the `"` and `\` characters, respectively; the backslash (`\`) (ASCII 92) character does not appear by itself in the string except in these two sequences. Embedded valid control characters (SPACE, TAB, LF) are preserved during processing. The empty string is written as `""`, eg.,

```
"This is a sentence."
"An embedded -> \" <- double-quote!"
"" ; (empty string)
```

2.5 Whitespace

Adjacent atoms are separated by whitespace: one or more SPACE, TAB, or LF characters. Whitespace is ignored between atoms, but is preserved in *string* atoms. No whitespace is required around a list delimiter.

2.6 Comments

Textual comments may be inserted anywhere where an atom or a list delimiter is expected – not within an atom. They are delimited by a semicolon (`;`) (ASCII 59) and the following LF; the characters between them are ignored, eg.,

```
(List ; this is a comment
      (Continued ; this is also a comment
            "List")) ; list ends here
```

3 GXF file structure

3.1 Introduction

GXF files encode hygraphs in a recursive list structure with the syntax specified in section 2. Since GXF files are meant to be used by very different processing programs, the only information that is *required* are data that specify the basic connectivity of a graph. Other information is optional, but processor programs may choose to support the visualization-related structures that **Hy+** uses.

Several kinds of structural errors can occur within a GXF file, for example, missing or multiple *string*s in a node-label structure, or reference to an unknown node-id. In general, a GXF processor program should cope with these errors by ignoring the structure in which the error occurred and continuing.

A GXF file consists of one or more GXF lists. At least one of these has the keyword `Graph`, and it contains the GXF graph as described below. Other lists may be included in a GXF file at its top level for use by processing programs (including other `Graph` lists).

The top-level `Graph` list is decomposed into sublists ("structures") that represent various parts of the hygraph: its edges, nodes, blobs, etc. These structures are decomposed further to describe individual edges, etc., and further yet to describe some individual data, such as label, position, etc. The required parts of this decomposition tree are specified by the GXF specifications.

Processor programs may extend this decomposition scheme by inserting additional data into a GXF sublist with (in context) unknown keyword, *number*, or *string* arguments within a known structure. An extension unaware processor program must ignore such arguments to insure that known GXF lists are handled as if the extensions were not there. Here is an extended node structure:

```
(Node
        (ID "1")
        (Label "'Toronto'")
        "Pink" ; This string is unexpected
        (Airport "Lester B. Pearson International"))
                ; This sublist is unexpected
```

A program that processes a GXF file should pass through to its output any extension information it finds, in order to preserve the information that is potentially useful to some other program.

The first few levels of the decomposition tree in a GXF file are shown in figure 3.1.

3.2 GXF structures

Each GXF structure has a keyword which is unique in the context of its containing lists. Starting with the "root", `Graph`, each structure is described below. Each description is of the form:

```
graph
    [graph-id]
    [graph-bounds]
    nodes*
        node*
            node-id
            node-label-ref | node-label
            [node-position]
            [node-bounds]
            [node-blobs-region]
            [node-icon-region]
            [node-label-region]
    edges*
        edge*
            from-node-id
            to-node-id
            edge-label
            [edge-points]
            [edge-distinguished]
    blobs*
        blob*
            blob-from-node-id
            blob-to-node-id
            blob-label
            [blob-bounds]
            [blob-nodes-region]
            [blob-label-region]
            [blob-distinguished]
```

Figure 1: Skeleton of GXF file structure. Each symbol refers to a GXF structure defined below.

name of structure `template of structure`

. . . description of **structure** and its arguments . . .

The arguments (other *components* or *atoms*) are explained in the description. Except where noted specifically, the arguments may occur in any order. Optional arguments are shown enclosed between `[` and `]`. Arguments that may be repeated zero or more times are shown suffixed by `*`. Alternatives are shown as { *one* | *two* }. All other arguments must be present as many times as listed.

The GXF structures follow.

---

graph (`Graph [`*graph-id*`] [`*graph-bounds*`]` *nodes*`*` *edges*`*` *blobs*`*)`

The **graph** list contains the definition of all components of a graph: its nodes, edges, blobs, etc. The graph's nodes are formed from the union of all *nodes* sublists of a *graph*, likewise for *edges*, *blobs* lists. Identification for the graph may be provided in *graph-id*. The default *graph-bounds* is implied from the *point* lists contained in the graph.

---

graph-id (`ID` *string*)

The **graph-id** list gives an identifier for the graph, coded as a GXF string, like this:

```
(ID "fche@samba.db:  Tue Jun 30 15:40:41 EDT 1992")
```

---

graph-bounds (`Bounds` *rectangle*)

The **graph-bounds** list specifies an optional bounding rectangle to the graph, possibly used to reserve space outside the "actual" contents of the graph.

---

rectangle (`Rectangle` *point point*)

The rectangle list specifies a two-dimensional rectangle. At least two *point* sublists are required, and specify two vertices of the rectangle.

---

nodes	`(Nodes` *node*`*)`
edges	`(Edges` *edge*`*)`
blobs	`(Blobs` *blob*`*)`

The **nodes** list contains set of sublists, each of which specifies a distinct node in the graph in which the list appears, just as the **edges** and **blobs** lists.

---

node	`(Node` *node-id* { *node-label* | *node-label-ref* } [*node-position*] [*node-bounds*] [*node-blobs-region*] [*node-icon-region*] [*node-label-region*]`)`

The **node** list defines a single node in the graph. The node is given a graphical region in one of several increasingly detailed ways, using the *node-position*; then the *node-bounds*; then the *node-blobs-region*, *node-icon-region*, and *node-label-region* sublists, respectively. The node is labeled directly by *node-label* or indirectly by node-label-ref.

---

node-id	`(ID` *string*`)`

The **node-id** list gives a node an identifying handle with which other objects (edges, blobs, etc.) can refer to the node. While it may be empty, *string* must be *unique* among all nodes.

---

node-label	`(Label` *string*`)`

The **node-label** list gives a node a textual label. While **Hy+** uses Prolog syntax (described in section 5) for node labels, these must be enclosed in GXF *string* delimiter double-quotes.

For example:

```
(Label "city('Toronto')")
```

---

node-label-ref	`(Label_Of` *string*`)`

The **node-label-ref** list gives a node an indirect label reference. This means that the label of the current node is the *identical to* that of the node identified by the node-id in *string*. The referenced node, defined somewhere else in the GXF file, must have a direct *node-label* label.

For example:

```
(Label_Of "node-10")
```

---

node-position (Point *point*)

The node-position list gives a node a single graphic location.

---

node-bounds (Bounds *rectangle*)

The node-bounds list gives a node a graphic region consisting of a bounding rectangle.

---

node-blobs-region (Blobs_region *rectangle*)

The node-blobs-region list gives a node a graphical bounding rectangle for the blobs it contains.

---

node-icon-region (Icon_region *rectangle*)

The node-icon-region list gives a node a graphical bounding rectangle for an icon representation.

---

node-label-region (Label_region *rectangle*)

The node-label-region list gives a node a graphical bounding rectangle for a text label representation.

---

point (XY *number number*)

The point list defines a single point in a two-dimensional co-ordinate system. The parameters give the X and Y co-ordinates, respectively. See section 3.3.

---

edge (Edge *from-node-id to-node-id edge-label* [*edge-points*] [*edge-distinguished*])

The edge list defines a single edge in the graph in which the list appears. Each of the specified sublists must occur at most once. By default, an edge has no points and is undistinguished.

---

from-node-id	`(From` *string*`)`
to-node-id	`(To` *string*`)`

The from-node-id and to-node-id lists give an edge a reference to its from-node and its to-node. Each argument is one of the node-ids assigned to a node in the graph.

---

edge-label	`(Label` *string*`)`

The edge-label list gives an edge a textual label. While **Hy+** uses Prolog syntax (described in section 5) for node labels, these must be enclosed in GXF *string* delimiter double-quotes. For example:

```
(Label "flight(ac,1000)")
```

---

edge-points	`(Points` *point**`)`

The edge-points list gives an edge a list of points that specify the path of the curve representing the edge from its from-node to its to-node. The curve points do not include the endpoints, and are taken in order of appearance. See section 3.3. Here is a curve with three middle-points:

```
(Points (XY 1 5) (XY 15 10) (XY 25 5))
```

---

edge-distinguished	`(Distinguished)`

The presence of an edge-distinguished list indicates that an edge is "distinguished".

---

blob	`(Blob` *blob-from-node-id blob-to-node-id** *blob-label* [*blob-bounds*] [*blob-nodes-region*] [*blob-label-region*] [*blob-distinguished*] `)`

The blob list defines a single blob in the graph (making it a *hygraph*). The blob is given a graphical region in one of several increasingly detailed ways, using the *blob-bounds*; then the *blob-nodes-region*, and *blob-label-region* sublists, respectively. The set of nodes contained by the blob is the set of those nodes whose node-ids are listed in any *blob-to-node-id* field. By default a blob is undistinguished.

---

blob-from-node-id (From *string*)
blob-to-node-id (To *string*)

The blob-from-node-id and blob-to-node-id lists give a blob a reference to additional nodes to be designated "contained" (to) or "container" (from), respectively. Each argument is one of the node-ids assigned to a node in the graph.

---

blob-label (Label *string*)

The blob-label list gives a blob a textual label. While **Hy+** uses Prolog syntax (described in section 5) for blob labels, these must be enclosed in GXF *string* delimiter double-quotes. For example:

```
(Label "cities")
```

---

blob-bounds (Bounds *rectangle*)

The *blob-bounds* list gives a blob a graphic region consisting of a bounding rectangle.

---

blob-nodes-region (Nodes_region *rectangle*)

The blob-nodes-region list gives a blob a graphical bounding rectangle for the nodes it contains.

---

blob-label-region (Label_region *rectangle*)

The blob-label-region list gives a blob a graphical bounding rectangle for a text label representation.

---

blob-distinguished (Distinguished)

The presence of a blob-distinguished list indicates that a blob is "distinguished".

3.3 Graphical co-ordinates

Some GXF structures associate points with nodes, edges, and blobs. These points can be used to display the graph. Since GXF, at the moment, transfers entire graphs, all points in the specified GXF fields are interpreted within the global `graph-bounds` co-ordinate system. With respect to the choice of co-ordinate system unit (pixels, cm, etc.), GXF imposes a certain anarchy: all co-ordinates in a graph are given in a single virtual co-ordinate system, within that graph's bounding rectangle (`graph-bounds`). If necessary, a processor program should scale and translate co-ordinates to its desired target co-ordinate system. A GXF reader program must adjust to handle *any* incoming co-ordinate system.

It is the responsibility of the GXF reader program to insure that co-ordinates given to graph components are valid. If an incoming graph has invalid layout (for example, a node-point outside its container blob's blob-bounds), and the reader program intends to use some of that information, the program must attempt necessary transformations to make the layout legal for the application (move the node to somewhere within the blob). Generally, the reader program should not rely on the correctness of the incoming layout.

4 GXF samples

4.1 A sample GXF file

In order to illustrate a complete GXF description of a small graph, we will demonstrate parts of the process of reading a GXF file and creating a visual display by the **Hy+** program. The graph here is a subset of one of our demonstration graphs. It contains a database of cities (nodes) and flights (edges from city to city, labeled with an acronym of an airline name). This graph contains only simple positioning information (*node-position*, *node-bounds*, *blob-bounds*), so **Hy+** infers the missing information.

Hy+ begins by reading the header of the GXF file:

```
; Hygraph fileout

(GRAPH
        (ID "fche@rock.db:  Fri Jul 31 12:07:44 EDT 1992" )
        (BOUNDS
                (RECTANGLE
                        (XY 0 0 )
                        (XY 1 1 )))
```

⋮ continued

This graph has a graph-bounds, so it will be used later. The contents of the graph-id field is shown to the user.

Next, **Hy+** reads the nodes of this graph:

⋮

```
        (NODES
                (NODE
                        (ID "1" )
                        (LABEL "'Hong_Kong'" )
                        (POINT
                                (XY 0.00903614 0.111111 )))
                (NODE
                        (ID "2" )
                        (LABEL "'Edmonton'" )
                        (POINT
                                (XY 0.454819 0.616162 )))
```

⋮ more nodes

```
                (NODE
                        (ID "6" )
                        (LABEL "'Canada'" )
                        (POINT
                                (XY 0.448795 0.111111 ))
                        (BOUNDS
                                (RECTANGLE
                                        (XY 0.448795 0.111111 )
                                        (XY 0.75 0.757576 ))))
```

⋮ more nodes

```
                (NODE
                        (ID "36" )
                        (LABEL "'Ottawa'" )
                        (POINT
                                (XY 0.689759 0.515151 ))))
```

⋮ continued

Next, **Hy+** reads the blobs:

⋮

```
        (BLOBS
                (BLOB
                        (FROM "6" )
                        (TO "2" )
                        (TO "9" )
                        (TO "25" )
                        (TO "33" )
                        (TO "3" )
                        (TO "36" )
                        (TO "7" )
                        (TO "26" )
                        (TO "31" )
                        (TO "23" )
                        (LABEL "cities" )
                        (BOUNDS
                                (RECTANGLE
                                        (XY 0.451807 0.161616 )
```

```
                                        (XY 0.746988 0.747475 ))))
```

⋮ more blobs

⋮ continued

Last, **Hy+** reads the edges:

```
⋮
        (EDGES
                (EDGE
                        (FROM "9" )
                        (TO "7" )
                        (LABEL "ac" ))
```

⋮ more edges

```
                (EDGE
                        (FROM "9" )
                        (TO "29" )
                        (LABEL "cp" ))
                (EDGE
                        (FROM "28" )
                        (TO "15" )
                        (LABEL "aa" ))))
; nodes:  36
; edges:  103
; blobs:  6
```

Hy+ has read all 36 nodes, 6 blobs, and 103 edges in this hygraph. If some edges had multiple segments, they would be drawn as polylines. **Hy+** can now verify the **graph-bounds**, fits the hygraph in its window with some margin around the outside, then displays the result as shown in figure 2. The finished graph window is ready to let the user browse, edit, and query the hygraph.

See figures 3 and 4 for example outputs of other programs equipped to read the GXF format.

4.2 Internet GXF archive

An anonymous `ftp` archive is available on the internet host `db.toronto.edu [128.100.2.100]` where files related to GXF may be obtained. Some of the GXF graphs used by **Hy+**, and other related information are available under the directory `pub/gplus/gxf`. You may contribute graphs of your own.

Also available at the `ftp` archive is a `C++` language programming library that includes a number of functions and data structures that can be used to create, manipulate, store, and retrieve GXF data. Please see the `README` files at the archive for further information.

5 Hy+ requirements

5.1 Introduction

GXF allows considerable freedom in using its defined structures to store general hygraph-encoded

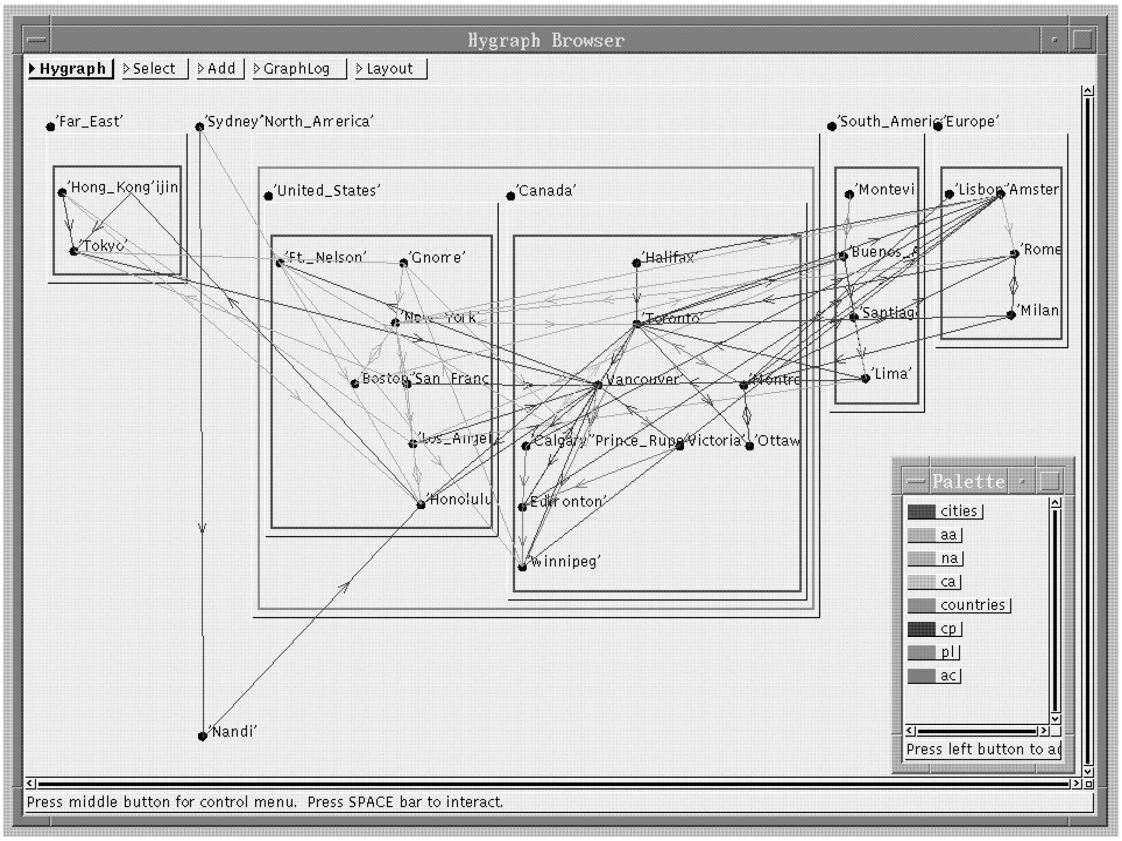

Figure 2: **Hy+** image of GXF graph, after all nodes, blobs, and edges have been read. Note that labels have been hidden to reduce screen clutter.

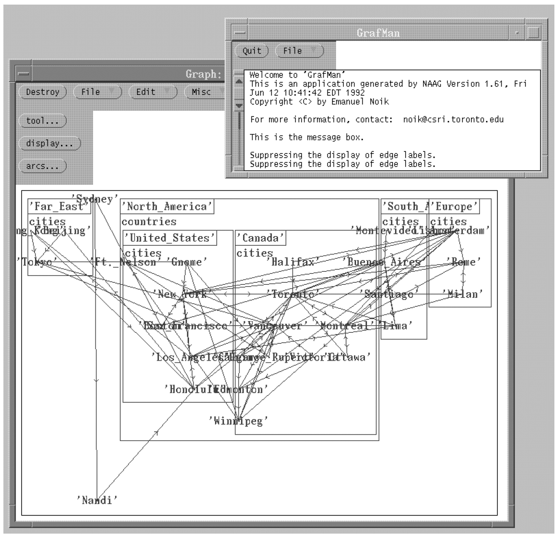

Figure 3: Display of Grafman (another graph browser/editor), reading the same GXF graph.

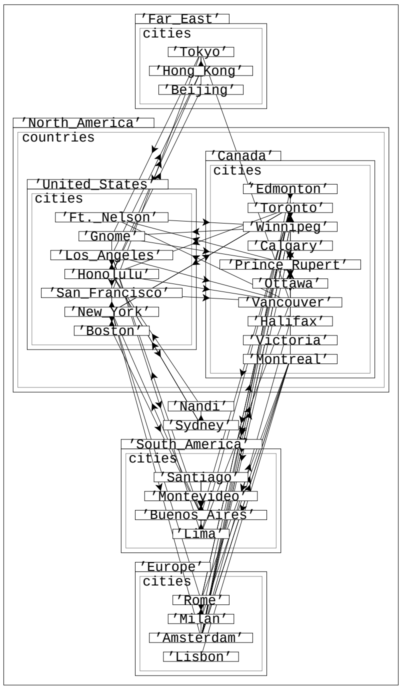

Figure 4: Output of a GXF-to-PostScript converter program, operating on a slightly laid-out copy of the same GXF graph.

data. **Hy+** can *visualize* any such data, but in order to *query* such data, the data must be in a more restricted form. While **Hy+** is robust with respect to most inconsistencies in a GXF file (e.g., incorrect node-id referencing, illegal layout, etc.), **Hy+** requires that all node, edge, and blob label strings obey corresponding GraphLog syntax [2], summarized formally below, using notation from section 2.3.

5.2 Labels in databases

node-label	:=	[node-label-part { `','` node-label-part }]
node-label-part	:=	functor-name `'('` functor-positions `')'`
	|	constant
functor-name	:=	string-constant-atom
functor-positions	:=	node-label-part { `','` node-label-part }
constant:	:=	string-constant-atom
	|	number-atom
edge-label	:=	predicate-name [`'('` edge-label-parts `')'`]
edge-label-parts	:=	edge-label-part { `','` edge-label-part }
edge-label-part	:=	node-label-part
blob-label	:=	edge-label

Here, the `number-atom` and `string-constant-atom` refer to syntactic atoms that obey standard Prolog syntax [5].

Some examples of each follow:

node-label	`'Toronto'`
	`city('Ottawa')`
	`city('Montreal'),price(10)`
edge-label	`flight`
	`flight(ac,price(100),class(first))`
blob-label	`cities`
	`countries('South')`

5.3 Labels in queries

Object labels in query graphs are more complex than those in a database graph. This is because a **Hy+** query can specify a regular expression, variables, arithmetic functions and aggregates, which are more complicated to describe. In addition, different rules apply to "distinguished" edges or blobs (see section 3.2) than to "undistinguished" ones because of a special interpretation by **Hy+** . Because a complete grammar for object labels in query graphs is rather long and of limited use, it is not specified here. (Contact the author if you need this information.)

References

[1] Mariano Consens and Masum Hasan. Supporting network management through declaratively specified data visualizations. To appear, 1993 IFIP/IEEE International Symposium in Network Management, available from the authors, 1992.

[2] Mariano Consens and Alberto Mendelzon. GraphLog: a visual formalism for real life recursion. In *Proceedings of the Ninth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1990.

[3] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *Proceedings of the 1991 CAS Conference*, October 1991. Also available as IBM Canada Laboratory Technical Report TR 74.053.

[4] Mariano P. Consens. Visual manipulation of database visualizations. PhD Thesis Proposal, in preparation, 1992.

[5] Richard J. Young. *Practical Prolog.* Van Nostrand Reinhold, 1988.