PIRL—Pattern Information Retrieval Language —Design of Syntax[*]

Dr. Sidney Berkowitz, Naval Ship Research & Development Center,** Washington, D. C.

The design of a pattern manipulation language, PIRL, is described here. PIRL can handle arbitrary, oriented patterns (i.e., subgraphs) of lists, nodes, numeric and Hollerith data on many levels of abstraction in a concise, legible manner. Patterns and lists may be inserted, retrieved, deleted, indexed, compared, named, intersected, united, and complemented. Pattern names may be referenced to a lower level of abstraction and pattern forms may be quantified. PIRL should be of considerable value in the solution of certain problems in information retrieval, linguistic analysis, scheduling simulation, and pattern recognition.

KEY WORDS AND PHRASES: graph, programming language, information retrieval, pattern, list attribute, association
CR CATEGORIES: 3.42, 3.60, 3.70, 3.81, 3.82, 4.22, 5.32

* The work was done under Task Area SF14 532 107, Task 15329.
**Computation and Mathematics Dept.

INTRODUCTION

PIRL (Pattern Information Retrieval Language) is a programming language designed to conveniently manipulate information in graph structures. As such, the language will play a key role in the construction of the organizational schemes found, for example, in information retrieval, linguistic analysis, and process scheduling systems. The language is written to complement an algebraic language such as FORTRAN or ALGOL, in the sense that PIRL statements are distinguished from the statements of the algebraic language and may be interleaved with those statements. The primary advantage of separating symbolic and numeric statements is that the programmer is afforded a linear, one-one trace of graph operations in the code description. From an opposing point of view, Feldman and Rovner's LEAP [1] and Ross's AED-0 [2], for example, are extensions of ALGOL in the sense that graph or list operations are interspersed with numeric operations. The result is that code sequencing of graph operations is bound by the infix, phrase-substitution nature of the algebraic language, and does not lend itself to an easy scan of the graph. On the other hand, the ALGOL extensions offer a uniformity of notation necessarily missing from PIRL.

The function of PIRL is to identify, insert, retrieve, delete, and compare subgraphs, or pattern images, which range from single nodes and lists to arbitrary, directed graph structures. As such, PIRL is an expansion of a less sophisticated language called GIRL (Graph Information Retrieval Language), reported elsewhere [3], which confines its graph manipulation activities to single nodes and lists. The GIRL language was designed to be primitive enough to allow the bootstrapping of the PIRL compiler (or preprocessor, as happens to be the case) from the GIRL preprocessor, in terms of GIRL statements. Feldman and Rovner's paper [1] contains a discussion and comprehensive bibliography of the languages and programming systems which lie in the background of current thinking on associative languages [4, 5, 6, 7].

After giving a very brief description of GIRL, this paper will discuss the design considerations behind PIRL. This paper is not intended as a programming manual, but rather as a description of the pros and cons of a language design. A concise rendering of the GIRL syntax may be found in the Appendix.

NOTATION OF GIRL

The function of GIRL is to manipulate node-link-node structures of the type shown in Figure 1. One may think of such a structure as a function (B), argument (A), value (C) triplet; or as a subject (A), relation (B), object (C) association. The language is based on the operations given in Table 1.

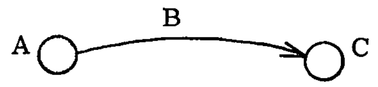


FIGURE 1—Node-link-node triplet

From a description as brief as the current one, the reader cannot hope to achieve programming competence in the language, but the following comments, together with the example, should give an idea of the basic architecture of the language.

Note first that insertion (non-selective) is non-destructive, so that links may be considered as multivalued functions, and sink nodes as ordered sets (or lists). Secondly, the X in Table 1 may be replaced by any left-hand portion—i. e., prefix—of a GIRL statement whose value is a node when processed in a left-to-right scan. Accordingly, parenthesization is employed to indicate a sequence of suffixes of a GIRL statement and imbedding occurs strictly to the right, as opposed to the phrase substitution function of algebraic infix notation for which imbedding is not necessarily oriented. A notion of "prefix sequence" was considered but discarded due to difficulties in legibility and compilation. In fact, the reader will see that the notion of a pattern to be introduced later will obviate the need for a prefix sequence. Thirdly, the conditional transfer sequence may test the result of any operation (except definition) and continue to process the statement when 1) the transfer

TABLE 1—Operations in GIRL

Form	Function	Meaning
X Y Z	Insertion	Connect source node X to sink node Z by link Y.
X Y. K C	Selective insertion	Let the sink node list linked to X by Y contain as its Kth item the node C.
X+Y	Retrieval	Find the node(s) linked to node X by the link Y.
X-Y	Deletion	Let X no longer be associated with any node(s) by the link Y.
X=Y	Comparison	Do X and Y refer to the same node?
X+Y. I	Indexing	Find the Ith item on the list accessed by X+Y.
X≡Y	Node definition	Give the name Y to the node (link, list) referred to by X.
/F/S	Conditional transfer	If the preceding operation fails, go to the label F; otherwise to S.

is written as F and success occurs, or
2) the transfer is written as //S and failure occurs. Note that failure of insertion occurs when the node-link-node triplet has already been inserted. The following example should help clarify the preceding notation.

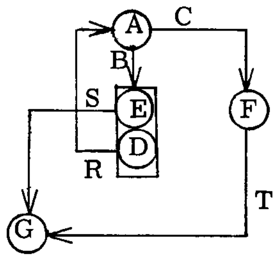


FIGURE 2—Sample graph

Suppose the graph shown in Figure 2 has been inserted. Then the following GIRL statement:

G 5 A≡X+B. 2(S F M N, +R(-C, =X/5))

produces the graph shown in Figure 3.

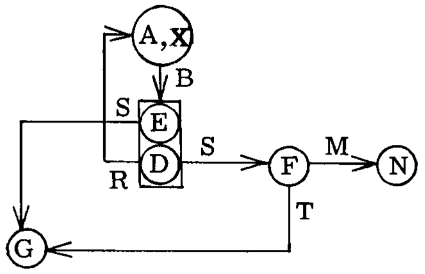


FIGURE 3—Sample graph transformed

The statement will have the following interpretation:

G	The statement is a GIRL statement.
5	The statement has label 5.
A≡X	Let A have the name X.
+B	From X, follow the B link to the list E, D.
. 2	Take the second item on the list E, D; i. e., D.
(S F	Insert the association "D linked by S to F".
M N	Insert the association "F linked by M to N".
, +R	(Second suffix) From D, follow the R link (to A).
, -C	If A has a C link, delete it.
, =X/5	(Second suffix) Do A and X refer to the same node? If not return to 5; otherwise continue.

In addition to the above operations, and aside from many details omitted for the sake of brevity, one should be aware of GIRL facilities for identifier definition, numeric and Hollerith data, and function subprograms.

· Identifier definition. There are two equivalent ways to assign internal nodes to identifiers, say X1, Y1: either by writing the GIRL statement

G DEFINE(X1, Y1)

or by writing $≡X1 and $≡Y1, either as separate statements or within the context of a statement, where $ means: generate a random internal node address. The identifiers then are variables having integer values which can be accessed in algebraic statements.

· Numeric and Hollerith data. One can insert data as a sink node or list in the form of 1) numbers, 2) the value of algebraic expressions, or 3) Hollerith strings. For example, X G("/5/3+X2Y", "IA+2", "I4") means: insert the character string 3+X2Y (of length 5), the value of the integer expression A+2, and the integer 4, respectively, as list values of the function G of X.

· Function subprograms. One can replace a node or link identifier at any point in a GIRL statement by a function of the form *ID(A1, A2,..., AN), where ID is an identifier and (A1, A2,..., AN) is a list of input identifiers, algebraic variables, or constants. The function is defined in the same way as a procedure of the complementary algebraic language, and may contain GIRL statements. Whether or not the function is recursive depends on the recursive capabilities of the algebraic language.

STRUCTURE OF PIRL

Patterns and Pattern Forms

In designing PIRL, the main objective was to produce a concise notation for the identification and retrieval of subgraphs (or patterns) contained within a stored graph. Moreover, the notation was to be compatible with the syntax of GIRL so that patterns could be manipulated as generalized nodes in a manner to be discussed. Since the retrieval of patterns in no way transforms any of the underlying graph structure, use of a bracketed GIRL statement which does not contain any insertions or deletions seems an appropriate way of describing a pattern. For example, the sample graph of Figure 2 might be retrieved as the pattern expression in (1).

<A+(C=F/1+T=G/2, B(. 1=E/3+S=G/4, . 2=D/5+R=A/6))> (1)

The transfers and comparisons are optional, and one can name the entire bracketed expression in the same way as one names a node. In the example just given, the node identifiers are assumed to have been defined by a DEFINE statement or definition operator. If the identifiers were not defined, they would be undefined, or free. The resulting bracketed statement would then be a pattern form and would refer to the list of patterns generated by the pattern form as the free identifiers range over the nodes of the graph. Thus, a free identifier is a symbolic variable, and a defined identifier is a symbolic constant. The implications of the preceding choice of notation for patterns and pattern forms within statements produce a host of problems which will now be discussed in a not-too-random order.

Transformation of Levels of Abstraction

If patterns (or their external names) are to be used in place of nodes, one must decide whether the list of internal addresses of the nodes contained in the pattern or the address of the pattern name is to replace the pattern (or its external name). If, during execution, one were to substitute the address of the pattern name for a pattern, then all pattern linking would occur among names. Consequently, in order to process the graph at the node level and determine the existing linkage, one would require a means of interrogating a node to discover with which patterns—one level of abstraction up—the node was associated. The node retrieval time would be greater than GIRL retrieval time; storage would be required both for name-to-pattern retrieval and for the inverse retrieval (although it is true that no storage would be needed for pattern node linkage); and, most important, association between patterns would be confined to the effective association of all nodes of one pattern with all nodes of another by a common link. It is preferable, then, to introduce the effective naming scheme shown in Figure 4, and a unary reference operator $\downarrow$ to descend from a name to its referent pattern. In other words, the occurrence of a pattern in a statement is replaced at execution time by its internal name; but if the pattern itself is desired, a reference operation can access the pattern. Storage is now required for pattern node linkage, but no level ascension is needed, and node retrieval is the same as in GIRL. Moreover, one can handle unconnected subgraphs as components of a single pattern. As an example of this structure, consider Figure 4. In order to

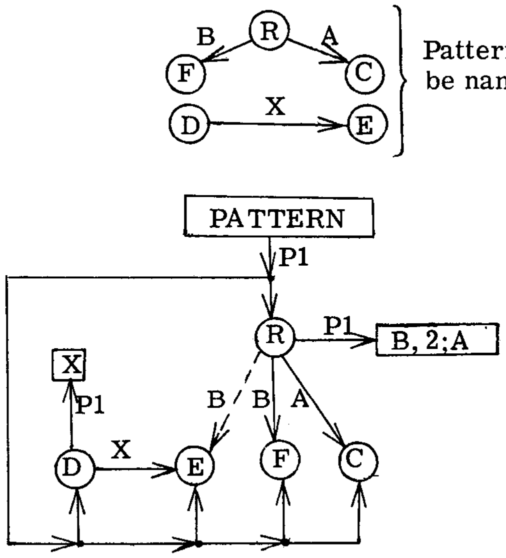


FIGURE 4—Example of pattern naming organization

avoid ambiguity, the nodes PATTERN and P1 are, in fact, internally generated, distinguished, random nodes not otherwise used in the graph. The PATTERN node is a source for all pattern names and each pattern links all of its nodes to PATTERN node, using the pattern name (e. g., P1) as link. Moreover, in order to avoid retrieval ambiguity, the P1 link from a pattern node R points both to the subset of links of the graph which are intended as links in the pattern from R, and also to the subset of indices for each multivalued link. That is to say, if one specified only the link subset, then links in the graph not to be included in the pattern—such as the dashed B link in Figure 4—would spuriously form a part of the pattern. Note that simply linking the nodes to the PATTERN node does not unambiguously exclude links of the type just mentioned, since the linked nodes may enter the pattern by linkage outside the pattern, as does E in Figure 4. Note also that linking all pattern nodes by the name link is not as redundant as it would seem, since the linkage not only provides unambiguous termination of loop traces, but also saves much time in pattern node retrieval as against a straightforward trace of the pattern. Since a similar—but simpler—structure obtains for lists (see Figure 5), one can speak of lists of pattern names (or patterns of list names) to any level of abstraction. Indeed, a consistent extension of the notation defined earlier for patterns might, for example, have the form:

<A, B, C, D>

where each argument represents a string whose value is an internal address. Such a list form may be named, and the subgraph underlying the abstraction is retrievable by

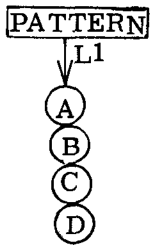


FIGURE 5—Example of list naming organization

successive reference operations. One byproduct of the naming scheme is that a name refers effectively to an internal node address uniformly for nodes, lists, or patterns, so that a special naming notation is not needed to distinguish nodes from more complex structures, as was previously thought [3].

Now that lists have been structured as types of patterns, one wonders whether it would not be consistent to structure nodes as types of lists. If we suppose that a node and its name were not distinguishable, then an insertion would require knowing whether a list, a node, or nothing had been inserted. Insertion would then take place by adding to the list, creating a new list, or creating a node-link-node structure,respectively. For example, if one wished to execute A B C, it would be necessary to replace the insertion triplet with the following GIRL code at compilation time:

```
G    A(+B≡D//1, B C //3)
G 1  PATTERN(+D/2, D C//3)
G 2  A (-B, B$≡NEW)
G     PATTERN NEW(D, C)
  3
```

As an alternative to this cumbersome interpretation of insertion, one might regard a sink node as a univalue list, so that the insertion of a new node-link-node triplet would require a list name definition. Thus, the insertion triplet A B C would be replaced at compilation time by the code:

```
G    A(+B≡D//1, B $≡D)
G 1  PATTERN D C
```

On the other hand, this procedure would require our writing A+B↓ when the node, not the node name, was desired. Although this is not too great a demand, one might avoid even this requirement (and extra storage, by the way) by replacing A+B by the sink node C, when A+B points to a strictly univalue list. Thus, A+B would be replaced at compilation time by the code:

```
G     A+B≡C/FAIL
G     PATTERN+C(.2//1, .1≡C)
  1
```

This scheme seems an inequitable trade, however, for the burden of using a reference operator. Thus, we have decided to regard sink nodes as univalue lists, and retrieval as the retrieval of a list name. On the basis of this decision, note that the pattern definition in Expression (1) above requires the insertion of a reference operator after every retrieval.

By introducing a reference operator, one makes it possible to substitute an internal address for a pattern at execution time, as stated above. But what substitution should one make for an external name? If one were to substitute the pattern (or list) itself, one would need a different reference operator, say ↑, to access the internal address of the name, one level up. On the other hand, if one were to substitute the internal address, one would at times need a formalism like [P↓] to access the pattern (or list) referred to by P; that is, one would need brackets so that the reference operation would not be applied to the value of the string preceding P. In fact, despite the possibility of more frequent use of the pattern than of its name, we have chosen the latter course, since the ↑ operation 1) would not be applied to bracketed expressions, 2) would not have a unique value, 3) would be costly in terms of storage and implementation time, and 4) would, in any case, require brackets for the same reason that the ↓ operation does.

In passing, one might mention a third organization that could prove feasible if translation occurred in an interpretive mode (not currently the case): namely, the representation of a pattern as a subroutine which would generate the pattern nodes and links, and which would be constructed dynamically as the execution of a statement produced and named new patterns.

Operations on Patterns and Lists

The attempt to replace nodes or links in statements by patterns or lists opens the way to several interpretations of the operations given above in Table 1. For example, if P1, L, and P2 are pattern names, should [P1↓][L↓] [P2↓] mean that each node of P1 should be linked with each node of P2 by the node addresses of L, a complete linkage, as shown in Figure 6, or that the first node of P1 is linked to the first one of P2, the next to the next, etc, a mirror linkage as shown in Figure 7. Or, perhaps one should introduce new notation to permute and reduce the source and sink lists? Since the last two possibilities require a detailed knowledge of the ordering of the pattern contents—not the usual circumstance—it is reasonable to give [P1↓] [L↓] [P2↓] the first interpretation, which is equivalent to regarding lists as unordered sets. For mirror linkage, we will

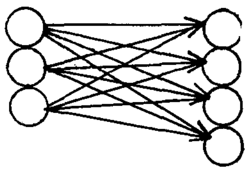
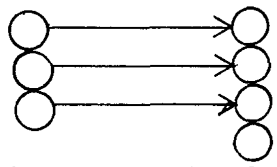


FIGURE 6—Complete linkage

use the notation ":" (e. g., [P1↓]:[L↓] [P2↓]). For permutation and reduction, careful definition or redefinition of patterns should suffice, and no new notation will be introduced.



(Each link represents a link list)

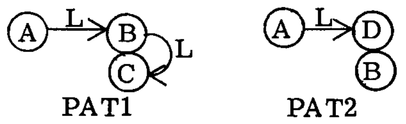
FIGURE 7—Mirror linkage

Another problem arises in interpreting the expression P1↓+[P2↓]. Certainly the value is a list. But should the list itemize nodes and list-names? Each name would refer to a sink node list retrieved by applying [P2↓]to a particular node in P1. If such were the case, one would, for consistency's sake, require the expression A+B↓↓ in order to access a sink node. Therefore, in order to avoid a cumbersome notation in the case of node retrieval, it is sufficient to interpret the original expression as producing an unstructured list of all the nodes retrieved.

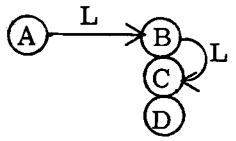
Still another problem arises with the expression P1↓≡A. If P1↓ produced a node, should A be given the value of the node address or that of the name address? The issue is decided by noting that if P1↓ were a list, one would be forced to say that identifier definition must be definition of the list. But then how would one identify a node? The solution is given by use of the index operation: i. e., P1. 1≡A gives to A the value of the first (and perhaps only) node address in P1.

In addition to basic GIRL operations, PIRL contains operations necessary for the manipulation of sets structured as patterns and lists, namely: ∪ (union), ∩ (intersection), ¬ (complementation). The union and intersection operations have the usual connotations of joining and excluding, respectively, those components which are held in common, as Figure 8 shows. On the other hand, complementation here is a binary operation and serves to exclude a pattern, node, or list from the preceding argument, as shown in Figure 8; that is, [P↓]¬[R↓] produces the complement of R relative to P. This peculiar definition arises from a reluctance to suffer the tedious implementation of a universal complementation (that is, a complementation relative to the whole graph). Finally, note that an operation may result in a null pattern or list, represented internally by a pattern or list name with no further linkage.

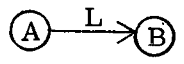
(a) pattern definitions:



(b) union: PAT1$\downarrow$∪[PAT2$\downarrow$]



(c) intersection: PAT1 ∩[PAT2$\downarrow$]



(d) complementation: PAT1$\downarrow$¬[PAT1$\downarrow$∩ [PAT2$\downarrow$]]

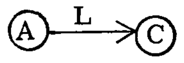


FIGURE 8—Set operations

Now that set operations have been introduced, it seems reasonable to use them—and indeed, any operation discussed thus far—to concatenate patterns, i. e., bracketed expressions. The corollary of this decision is to allow nested brackets in PIRL statements,an infix notation which neutralizes the vaunted legibility of the left-to-right scan of GIRL statements. For the sake of legibility, one could of course insist on manipulating bracketed expressions outside of GIRL statements, naming the result, and using only names in GIRL statements, but this hardly seems a sufficient reason to exclude the convenience of mixed notation. Accordingly, we also permit the stand-alone bracketing of indexed, referenced, or redefined identifiers (e. g., [R. 1], [R$\downarrow$], [R≡S]), or combinations thereof. Also, conditional transfers should be permitted within such brackets.

The preceding discussion by no means covers all the design issues involved, but does give a fair notion of the outstanding problems.

Quantification and Logical Operations

The use of free identifiers permits one to find pattern instances anywhere in the stored graph. For the sake of descriptive convenience, it may be desirable to limit the search to a specified subgraph. One should note that such a limitation may decrease the retrieval efficiency, especially if one uses a paging system such as that in LEAP[1]. One means of specifying search limits would be to introduce a sequence of quantifiers, such as (for) ALL and (there) EXIST , each bounding a succeeding sequence of free identifiers found in a pattern form to follow. The EXIST quantifier might be followed, optionally, by the number of patterns expected to be found. The entire expression, when evaluated, would produce a name of a list of pattern names. For compilation ease, and indeed for user convenience in remembering which identifiers are free, the use of free identifiers is restricted to quantified pattern forms (or, as we shall see, to a logical concatenation of quantified pattern forms). Although the definition of a free identifier is local to the range of its bounding quantifier, the free identifier may be renamed as a global identifier within a pattern form. The new name is used to hold the internal name of a list of values which successfully match the pattern form in the free identifier.

One might further delimit the search by introducing a notation for membership— say IN P, where P is a pattern name—to precede the pattern form to be quantified. One might alternatively, or additionally, consider replacing the pattern P by an expression, say {G, d, L}, which would limit the search to nodes emanating from a node G, to a depth d, by links contained on a list L. Moreover, one might introduce the logical connectives AND, OR, NOT to concatenate quantified forms. However, the implementation of these notations, appealing as they may be, is quite elaborate. Therefore, in the first phase implementation, we restrict ourselves to logical connectives and two quantifiers before the outermost bracket of a pattern form, and leave membership and more complex quantification for another time.

A fundamental objective in creating the notations of pattern and pattern form was to allow for the sequence:

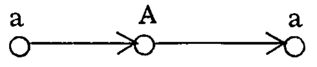
quantified ⟨pattern match⟩ ⟶
pattern transformation or further match. (2)

Like FORTRAN, PIRL has conditional transfers to accomplish the transition between matching—say, an algebraic condition in the case of FORTRAN—and transformation, or even further matching. On the other hand, ALGOL provides an <u>if</u>. . . , <u>then</u>. . . , <u>else</u>. . . structure that has some appeal, from the viewpoint of descriptive power: i. e. the capability of coding a potentially large subgraph of a flowchart in one statement. As a by-product, free identifiers which are evaluated in a pattern match would be available for use in the transformation field without naming. Nevertheless, one must allow in any case for the naming of free variables (i. e. the naming of the list of values which satisfy the pattern form) so that transformations under the control of conditional transfers may use the names. In the interest of minimizing the notation explosion and keeping statements concise and legible, we adopt a compromise between conditional transfers and <u>if</u>. . . , <u>then</u>. . . , <u>else</u>. . . , by introducing the operator ⟶ as above in Equation (2), and by requiring free identifiers to be renamed for global usage. Thus, the else field is effectively placed on another line and is executed by the failure option of a conditional transfer following the pattern match. The example which follows should clarify these ideas.

EXAMPLE

The example presented here is a parser for the web grammar representing the class of non-trival, basic, two-terminal, series-parallel networks (TTSPN's) discussed somewhat more elegantly by Pfaltz and Rosenfeld [8].

A non-trival, basic TTSPN is defined (roughly) as follows. Let the terminal vocabulary be {a}, the non-terminal vocabulary {A}. The initial string is:



Then any basic TTSPN can be derived from the initial string by repeated application of the following rules:

R1 (series expansion);
1. Expand a node A to the edge $\overset{A}{\circ}\longrightarrow\overset{A}{\circ}$
2. Let any edge originally entering A now enter the first node of the new edge.
3. Let any edge originally leaving A now leave the second node of the new edge.

R2 (parallel expansion): $\begin{matrix}\circ A\\ \circ A\end{matrix}$
1. Expand a node A to the nodes
2. Let any edge originally entering A now enter both new nodes.
3. Let any edge originally leaving A now leave both new nodes.

R3 (termination):
1. Replace A by a.

In order to code a PIRL program to recognize whether or not a given graph is basic TTSPN, we establish the following conventions for inputting a graph:
1. The graph must be entered from the node B1 by an R link, and must provide an exit via an R link to node B2.
2. Every link must be designated as R. For every pair of nodes A, B connected by R from A to B, insert link L from B to A. All links are connected by R links to B2 and by L links to B1.

The PIRL program, complemented to FORTRAN, follows, with explanatory remarks. (n. b. the symbol ≠, not heretofore introduced, means simply, "is not equal to", posed as a question. The expression . -K means "eliminate the Kth node from the preceding list".)

```
P  SER  ..EXIST A2[<A2≡A2N(≠(B1, B2),
P        *+R≡A3)>..AND..NOT[<A3.2>..OR
P        *<A3↓+L.2>]]/PAR⟶
P        *A2N.1≡A2(-R,+L≡A4,-L)
          N=1
       1  M=1
P         A4.N/SER≡A4N+R≡A4R
P      2  A4R.M/5=A2/3/4
       3  M=M+1
          GO TO 2
P      4  A4N R.M A3 L A4N
       5  N=N+1
          GO TO 1
P  PAR  ..EXIST A2,A4[<A2(≠(B1, B2),
P        *+L≡A1.1≡A11,+R≡A3.1≡A31>
P        *..AND..NOT[<A1.2>..OR<A3.2>
P        *..OR<A11=A31>]..AND<A4≡A41
P        *(≠(A11, A2, A31, B1, B2),+L.1=A11,
P        *+R.1=A31)>..AND<A11+R↓∩[A3↓]
P        *=NULL>]/TEST
```

```
          K=0
          J=0
P         A31. 1≡A3
P         A11. 1≡A1
P      6  K=K+1
P         A1+R. K=A4/6
       7  J=J+1
P         A3+L. J=A4/7
P         A1+R. -K
P         A3+L. -J
P         A4(-R, -L)//SER
P TEST .. EXIST A6[<A6(≠(B1, B2), +L↓=B1,
          +R↓=B2>] /8/NO
       8  B1+R. 2/YES/NO
```

Program

The first pattern match is for series reduction: if there are non-entry/exit nodes A2, A3 linked by single-valued R, L links, respectively, then reduce the graph by eliminating the A2 to/from A3 linkage and replacing A2 by A3; if such A2, A3 do not exist, try a parallel reduction: if there are non-entry/exit nodes A2, A4 which are both linked by single-valued R, L links to the same non-entry/exit nodes A3, Al, respectively, and if A1, A2, A3, A4 are all distinct and A1 and A3 are not linked, then reduce the graph by removing A4, and again try a series reduction; if such A2, A4 do not exist, test whether or not the graph has been reduced to the initial string.

IMPLEMENTATION

PIRL is being prepared to complement FORTRAN on a recently acquired CDC 6700. The GIRL portion of the system currently runs on the CDC 6700. The system will be available in several options which will allow the programmer to tune the system to some extent. These options include: no-list (destructive insert), fixed-length list definition (so that lists are accessed as vectors in sequential storage for the sake of speed but at a loss of dynamic storage), no-paging (for a small associative graph in main memory).

SUMMARY

The design of a pattern manipulation language, PIRL, has been described. PIRL can handle arbitrary, oriented patterns (i. e. subgraphs) of lists, nodes, numeric and Hollerith data on many levels of abstraction in a concise, legible manner. Patterns and lists may be inserted, retrieved, deleted, indexed, compared, named, intersected, united, and complemented. Pattern names may be referenced to a lower level of abstraction, and pattern forms may be quantified.

APPENDIX - SYNTAX OF PIRL

In the following abbreviated syntax of PIRL, the usual BNF notation suffices for the metasyntactic symbols ::= (is defined to be) and | (exclusive or); a string of small Roman characters represents a syntactic category; and FORTRAN Hollerith characters form the terminal alphabet. A list of mnemonics for the syntatic categories (in alphabetical order), together with a brief functional description of each category, precede the syntax. Heavy underlines in the syntax indicate the PIRL additions to the GIRL syntax.

Mnemonic	Category Description
a	alphabet
ae	algebraic expression
ans	alphabet with no slash
bc	blank string or colon
bk	single blank
bks	blank (space) string
bo	binary operator
cons	constant
data	data (numeric or Hollerith)
define	definition of identifiers
digit	digit
dix	data index
dnode	data node
dpatu	data pattern—unparenthesized
dseq	data in sequential space
dsuff	data suffix
emp	empty
GIRL-like statement	statement without pattern match and/or quantification
h	Hollerith string
hns	Hollerith string—no slash
i	identification
id	identifier (or substitute)
idcfl	identifier and/or constant list
idf	identifier
idfl	identifier list
idsub	identifier substitute
int	integer
ivc	integer variable or constant
ix	index
ixnd	index—no delete
ixvc	index followed by integer

Mnemonic	Category Description (cont.)
	variable or constant
la	(suffix) list type-a
label	label
labi	label and/or identification
ladix	(suffix) list type-a entry following data index operator
lai	(suffix) list type-a entry following definition operator
laix	(suffix) list type-a entry following index operator
lb	(suffix) list type-b
lbe	(suffix) list type-b following identification
lbo	logical binary operator
lbon	logical binary operator or ..NOT
lc	(suffix) list type-c
lcd	(suffix) list type-c data entry
lce	(suffix) list type-c entry
lcix	(suffix) list type-c entry following index operator
lcixvc	(suffix) list type-c entry starting with index operator
link	link
na	node type-a
nabc	node type-a, -b, or -c
nb	node type-b
nc	node type-c
node	node
pat	pattern
patu	pattern—unparenthesized
pbo	pattern binary operator
PIRL statement	PIRL statement
pdsuff	pattern data suffix
pexp	pattern expression
pl	pattern list
pla	pattern (suffix) list type-a
plb	pattern (suffix) list type-b
plbdix	pattern (suffix) list type-b—data index
plbi	pattern (suffix) list type-b—identifier
plbix	pattern (suffix) list type-b—index
pna	pattern node type-a
pnab	pattern node type-a or -b
pnb	pattern node type-b
ppt	pattern match to pattern match or transformation
psuff	pattern suffix
ptrans	pattern transformation
q	quantifier
qf	quantification
qlog	quantificational (or simple pattern) logic item
qp	quantified (or unquantified) pattern
ro	reference operator
suff	suffix
suffbc	suffix following by blank or colon
suffne	suffix—not empty
ta	transfer address
ti	transfer and/or identification
type	(sequential data) type

SYNTAX

1. <u>null</u> and <u>blank</u>

emp	::= (empty category)
bk	::= (blank character)
bks	::= bk|bk bks

Note: Blank sequences(bks) may be used at will, except where required explicitly by the syntax. Moreover, even where required explicitly, they may be omitted if <u>bks</u> is preceded or followed by a delimiter.

2. <u>identifiers</u>, <u>functions</u>, <u>definitions</u>, <u>lablels</u>, <u>transfers</u>

idf	::= (alphanumeric identifier)
cons	::= (constant)
idcfl	::= idf| cons| idcfl, idcfl
idsub	::= idf| $| * idf(idcfl)|<u>pat</u>
define	::= DEFINE bks idfl
idfl	::= idf|idfl, idf
i	::= ≡idf | i≡ idf| emp
id	::= idsub i
label	::= idf| int
ta	::= label|//label|/label/label
labi	::= i bks label i| label i
ti	::= i ti|/labi|//labi|/labi/labi |emp

Note: An identification (i) may appear anywhere in a statement (except after an operator). A function (*idf(idcfl)) or a randomly generated address ($) may substitute for an identifier. The label (label) may be variable if the complementary language allows, and must take the form stipulated by the complementary language. A transfer may occur anyplace in a string, and affords an execution control switch which depends on the success or failure of the last operation (excluding identification). Example:

/F1/S1 means: go to F1 if failure; go to S1 if success

//S1 means: continue if failure; go

```
                to S1 if success
/F1   means: go to F1 if failure; con-
                tinue if success
```

3. Hollerith, numeric data

```
dig      ::= 0|1|2|3|4|5|6|7|8|9
int      ::= dig|int dig
ans      ::= dig|A|B|C|D|E|F|G|H|I|J|K
             |L|M|N|O|P|Q|R|S|T|U|V|W
             |X|Y|Z|+|-|. |bk|*|(|)|[|]|<
             |>|=|≠|≡|, |;|: |$|↑|↓|→|∩|∪|⌐|"
a        ::= ans|/
h        ::= a|h a
hns      ::= ans|hns ans
ae       ::= (boolean, complex, real
             single-precision, real
             double-precision expression)
data     ::= "type ae"|"//hns"|"/int/h"
type     ::= B|C|R|D|I|O
ivc      ::= idf|int
dseq     ::= ; ivc ; ivc
```

Note: Data may be regarded as terminal sink nodes of an insertion operation (cf. dsuff below). Hollerith data may be entered following "//as a string of characters excluding quote between quotes, or as a counted string including quote. Thus, "//NO QUOTE", or "/6/QUO"TE". Data of type boolean (B), complex (C), double-precision real (D), Hollerith (H), integer (I) or real (R) may be entered into sequentially accessible memory at a given address. Thus ;5;1"IX+3" places the integer value of X+3 into SEQ(5), where SEQ is a reserved block of memory. Similarly, ;3H('//NSL/, 1/3/SLA) places the strings NSL and SLA into memory beginning at SEQ(3). All data can be list named and accessed by index as computer-dependent output.

4. unparenthesized statements

```
bo       ::= +|-|=|≠|∩|∪|⌐
ro       ::= ↓| emp
ix       ::= .|.-|.≡|.-≡|.-/|.-"
dix      ::= ./|."|.-/|.-"
ixi      ::= .|.≡|./|."
ixvc     ::= ixi ivc ti|emp
bc       ::= bks|:
link     ::= id ti
suffne   ::= bc link ixvc bks id ro suff
             | bo link ro suff|ti ro suff|ix
             ivc ro suff|≡idf ro suff
suff     ::= suffne|emp
dsuff    ::= bc link ixvc dseq data ti
             |bc link ixvc data ti|dix ivc
             ti|suff dsuff
node     ::= id ro suffne|[node]
dnode    ::= id ro dsuff
nodei    ::= node|id
```

5. parenthesized statements

```
suffbc   ::= suffne bc|emp
nabc     ::= na|nb|nc|[nabc]
na       ::= nodei ro (la)
la       ::= suffne|dsuff|suff bo lbe
             | suff bo (lb)|suff ix (laix)
             | suff dix (ladix)|suff≡(lai)
             | suffbc link ixvc bks nabc
             | suffbc link ixvc (lc)
             | suffbc link ixnd (lcix)
             | suffbc link (lcixvc)|ro la
             | la, la
laix     ::= ivc la|laix, laix
ladix    ::= ivc ti|ladix, ladix
lai      ::= idf la|lai, lai
nb       ::= nodei ro bo (lb)|nodei≡(lai)
lb       ::= lbe|lb, lb
lbe      ::= link|node|dnode|nabc
nc       ::= nodei bc link ixvc (lc)
             |nodei bc link ixvc dseq (lcd)
             |nodei bc link ixi (lcix)
             |nodei bc link (lcixvc)
lc       ::= lce|ro lc|lc, lc
lce      ::= lbe|data ti
lcd      ::= data ti|lcd, lcd
lcix     ::= ivc ti dseq|ivc ti bks lce|lcix, lcix
lcixvc   ::= ixvc dseq data ti|ixvc bks
             lce|lcixvc, lcixvc
GIRL-like ::= node|dnode|nabc|define
statement     |ta|id≡idf
```

Note: An unparenthesized string (node, dnode) consists of a prefix string (id ro suff) followed by a suffix string (suffne) which may be continued or a data suffix (dsuff) which terminates the string. The suffix operates on the node address or pattern of addresses produced by the prefix string in a strict left-to-right scan. PIRL employs parentheses to sequence suffixes of an unparenthesized prefix pattern. Moreover, the unparenthesized portion of each suffix may itself be a prefix to a suffix sequence. Scanning and imbedding are strictly to the right. Thus, for example, <A+B>↓(C X, D Y) means: "find the pattern name associated with <A+B>, access the pattern itself and completely link the pattern to X by C and to D by Y.

6. quantified statements and pattern transformations

```
q        ::= ..ALL|..EXIST
lbo      ::= ..AND|..OR
lbon     ::= ..NOT|emp
qf       ::= q bks idfl|qf bks qf
qp       ::= pat|qf pat|qp ti|[qp]
qlog     ::= qp|qlog lbo lbon bks qlog
             |qlog lbo lbon[qlog]
             |qf [qlog]|[qlog]
ppt      ::= qlog|..NOT bks qlog
             |..NOT [qlog]|ppt→ppt
ptrans   ::= ppt→GIRL-like statement
```

7. unparenthesized pattern expressions

```
pbo      ::= +|=|≠|∩|∪|→
psuff    ::= pbo pat  psuff|ti psuff
             |≡idf psuff|. ivc psuff|emp
pdsuff   ::= dix ivc ti|psuff pdsuff
patu     ::= id ro psuff|[patu]
dpatu    ::= id ro pdsuff
pexp     ::= patu|dpatu
```

8. parenthesized patterns and pattern strings

```
pnab     ::= pna|pnb
pna      ::= patu (pla)
pla      ::= psuff|pdsuff|psuff pbo plb
             |psuff pbo (plbdix)|[pla]
             |psuff. (plbix)|psuff dix
             (plbdix)|psuff ≡ (plbi)|ro pla
             |pla, pla
pnb      ::= patu pbo (plb)|patu ≡ (plbi)
plb      ::= patu|dpatu|pnab|[plb]|plb, plb
plbix    ::= ivc pla|plbix, plbix
plbdix   ::= ivc ti|plbdix, plbdix
plbi     ::= idf pla|plbi, plbi
pl       ::= pexp|pnab|pat|pl, pl
pat      ::= <pl>|pat↓|[pat]
```

REFERENCES

1. Feldman, J. A. and Rovner, P. D., "An ALGOL-based associative language", Comm. ACM, 12, 8, pp. 439-449, 1969

2. Ross, D. T., "A generalized technique for symbol manipulation and numerical computation", Comm. ACM, 4, 3, pp. 147-150, 1961

3. Berkowitz, S., "Graph information retrieval language—design of syntax", Software Engineering (Proc. COINS-69), J. T. Tou (ed.), Vol. 2, pp. 119-139, 1971 Academic Press, New York

4. Christensen, C., "An example of the manipulation of graphs using the AMBIT/G programming language", Proc. Symp. Interactive Systems In Experim. Appl. Math., Washington, D.C., 1967

5. Knowlton, K. C., "A programmers description of L^6", Comm. ACM, 9, 8 p. 616, 1966

6. McCarthy, J., et al., Lisp 1.5 Programmer's Manual, MIT Press, Massachusetts, 1962

7. Newell, A. (Ed.), Information Processing Language-V Manual, Prentice-Hall, New Jersey, 1961

8. Pfaltz, J., and Rosenfeld, A., "Web grammars", Proc. of the International Joint Conference on Artificial Intelligence, p. 609, 1969