The Design of GRASPER 1.0: A Programming Language Extension for Graph Processing

John D. Lowrance and Daniel D. Corkill

COINS Technical Report 79-6

February 1979

The Design of GRASPER 1.0: A Programming Language Extension for Graph Processing

John D. Lowrance and Daniel D. Corkill

COINS Technical Report 79-6

February 1979

This work was supported by
The National Science Foundation
under grant number
MCS75-16098 A01.

The Design of GRASPER 1.0: A Programmiong Language Extension for Graph Processing

John D. Lowrance and Daniel D. Corkill

COINS Technical Report 79-6

Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

February 1979

ABSTRACT

GRASPER 1.0 is a programming language extension. Once appended to a host language, GRASPER 1.0 introduces graphs, diagrams consisting of points connected by lines or arrows, as a primitive data type.

The primary feature of GRASPER 1.0's design is that the language, its documentation, and its implementation all share a common organizational structure that groups GRASPER 1.0 primitives according to their scope of application and the underlying concepts from which they are formed. Although this report is of a descriptive nature, a similar approach might well be prescribed for other applications.

GRASPER 1.0 is based on a small number of underlying concepts. GRASPER 1.0 primitives are constructed from these concepts according to a small set of rules. The name of each GRASPER 1.0 primitive systematically reflects its underlying concepts. This generative nature of the language organizes a large set of primitives in a cognitively efficient way. This makes GRASPER 1.0 easier to learn and retain; provides an indexing system for GRASPER 1.0 documentation; and serves as an outline for well-structured implementations.

GRASPER 1.0 has been implemented with LISP 1.5 as the host language. This implementation supports a software-level virtual memory management system for graph storage. Spaces, user defined subgraphs, are used by the virtual memory manager to group logically related information on the same pages, helping to reduce paging. Multiple storage schemes allow users to optimize the way graphs are stored based on their particular applications.

ACKNOWLEDGEMENTS

We wish to thank Richard S. Brooks, Allen R. Hanson, Daryl T. Lawton, Edward M. Riseman, Janet E. Turnbull, Thomas D. Williams, and Bryant W. York for their contributions to this report.

This work was supported by The National Science Foundation under grant number MCS75-16098 A01.

TABLE OF CONTENTS

I. INTRODUCTION

Graphs, diagrams consisting of points connected by lines or arrows, are commonly used to depict situations of interest. GRASPER 1.0 is a programming language extension that provides graph processing capabilities. The ability to program directly in graph primitives is an obvious advantage in those areas where problems are naturally cast in graph terms (see Table 1).

GRASPER 1.0 was developed as a data base facility for the VISIONS system [HAN78a,b], a computer system for the segmentation and interpretation of visual scenes. VISIONS required that GRASPER 1.0 support large, dynamic graph structures. The design of GRASPER 1.0 (GRASPE Extended and Revised) is closely modeled after that of GRASPE 1.5 [PRA71]. GRASPER 1.0's formal foundation lies in set and graph theory. Informally, GRASPER 1.0 is based on the natural pictorial semantics of graphs.

GRASPER 1.0 consists of a set of operators that could potentially be appended to any list processing system. Once appended to a host language, GRASPER 1.0 introduces GRASPER-GRAPHs as a primitive data type (see Figures 1 and 2). GRASPER-GRAPHs consist of nodes, edges, and spaces[1]. Nodes, edges, and spaces all have names and values. Edges are directed connections between pairs of nodes. Spaces are

[1]GRASPER spaces, though similar, differ from those of Hendrix [HEN75].

GRAPH	FIELD
sociograms	psychology
simplexes	economics
state transition networks	automata theory
Markov chains	probability theory
PERT networks	management decisions
data structures	computer science
flow charts	chemistry, programming
crystal structures	physics
bonding structures	chemistry
transportation networks	operation research
family trees	genealogical theory
computer system configurations	computer architecture
semantic networks	artificial intelligence
augmented transition networks	artificial intelligence, linguistics
neural networks	neurophysiology, cybernetics
phrase markers	linguistics

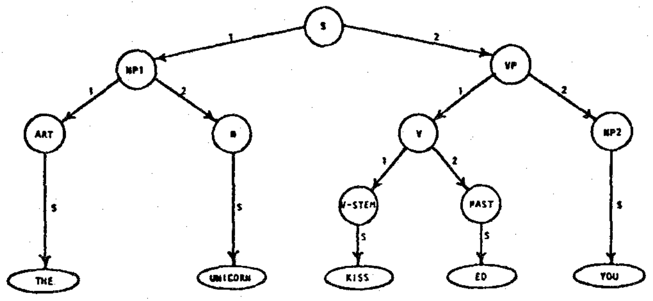
TABLE 1 - Examples of common graph applications

subsets of nodes, edges, and values, i.e., subgraphs. GRASPER-GRAPHs are created, queried, modified, and destroyed through GRASPER operators. The arguments to these operators, and their results, are lists and atomic elements of the host language.

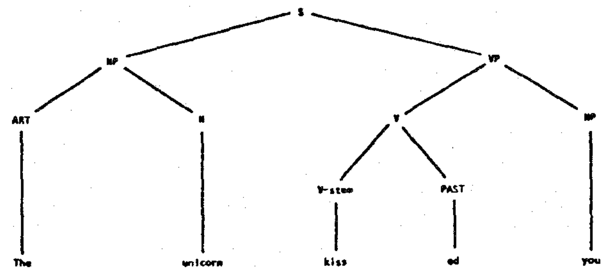
This report describes GRASPER 1.0, highlighting the features of its design. The primary design feature is that the language, its documentation, and its implementation all share a common organizational structure. Although this report is of a descriptive nature, a similar organizational approach might well be prescribed for other applications.

Throughout the remainder of this report "GRASPER 1.0" is abbreviated "GRASPER."

Phrase Marker

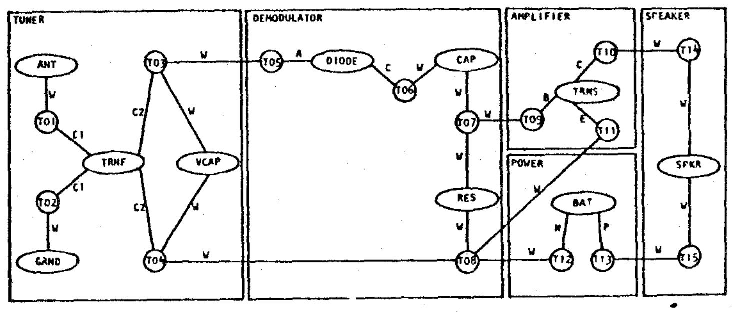


This GRAPH represents the following phrase marker.

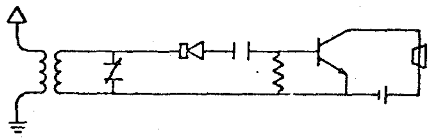


The direction and labeling of edges in the GRAPH are used to incorporate the information implicit in the position of the terminals and nonterminals in the phrase marker. All edges point (down) towards the surface. Numbers on edges order the nonterminals to which they point. S-edges point to surface terminals.

Radio



This GRAPH represents the following schematic for a radio.



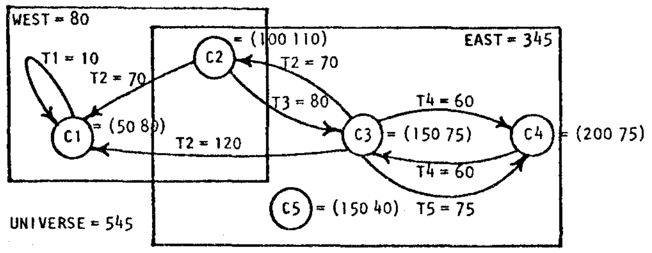
The labels on the GRAPH translate as follows.

ANT - antenna	RES - resistor
GRND - ground	TRNS - transistor
TRNF - transformer	SPKR - speaker
VCAP - variable capacitor	BAT - battery
DIODE - diode	T# - bread board terminal
CAP - capacitor	W - wire

The undirected edges indicate two edges labeled the same with opposite direction. The spaces indicate the major components of the radio.

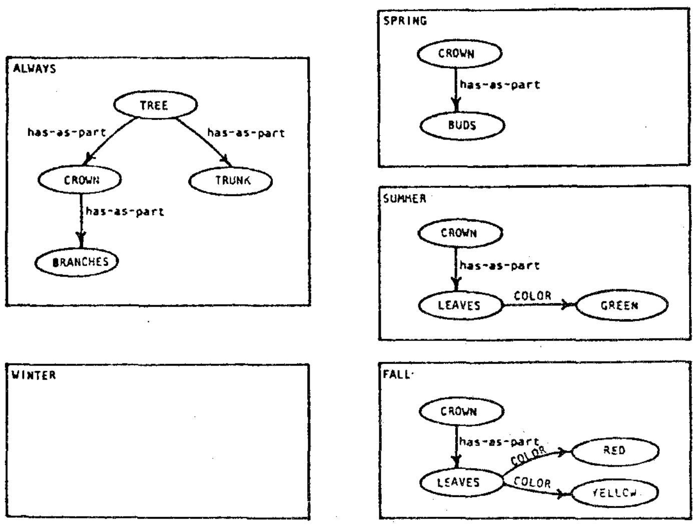
FIGURE 1 - Example GRASPER-GRAPHs

Railroad



The above GRAPH represents a railroad's system of tracks servicing five cities. Each city is represented by a node. Tracks are represented by edges between cities. Commuter tracks local to a city are represented as edges which originate and end at that city. The direction of each edge indicates the direction trains travel on that track during morning rush hour. The names of tracks correspond to routes. The universal value of each city indicates its cartesian coordinates within the railroad system. The universal values of the tracks indicate their length. The east and west division of the railroad are delimited by spaces. The value of each space is the total amount of track within it.

Tree



The above GRAPH is a semantic network describing trees over the seasons. The space ALWAYS contains information about trees that is true during all seasons. Each of the other spaces contains information about trees that is true in the corresponding season.

Note that unlike the previous illustrations, each space has been broken apart from the rest of the GRAPH. That is why some of the nodes and edges appear more than once (e.g., the node LEAVES and the edge HAS-AS-PART from node CROWN to node LEAVES appear in spaces SUMMER and FALL).

FIGURE 2 - Example GRASPER-GRAPHs

II. DESIGN APPROACH

A wholistic approach was taken in designing GRASPER. Design decisions were made only after considering their impact on all aspects of GRASPER: the language, its documentation, and its implementation. This approach centered on the development of an organizational structure that groups GRASPER primitives according to their scope of application and the underlying concepts from which they are formed (see Figure 3).

GRASPER primitives are divided into three groups according to their scope of application. Group I primitives apply to individual units of GRASPER-GRAPHs. Group II primitives apply to major portions of GRASPER-GRAPHs. Group III primitives control memory management for GRASPER-GRAPH storage. For example, it is Group I primitives that create individual nodes and edges, Group II primitives that create entire graphs, and Group III primitives that move graphs between short-term and long-term storage.

All GRASPER primitives are based on a small number of underlying concepts. They are constructed from these concepts via a small set of composition rules. The name of each primitive systematically reflects the underlying concepts from which it is formed. This generative nature of the language, along with the grouping of primitives by scope, organizes a large set of primitives in a cognitively efficient way.

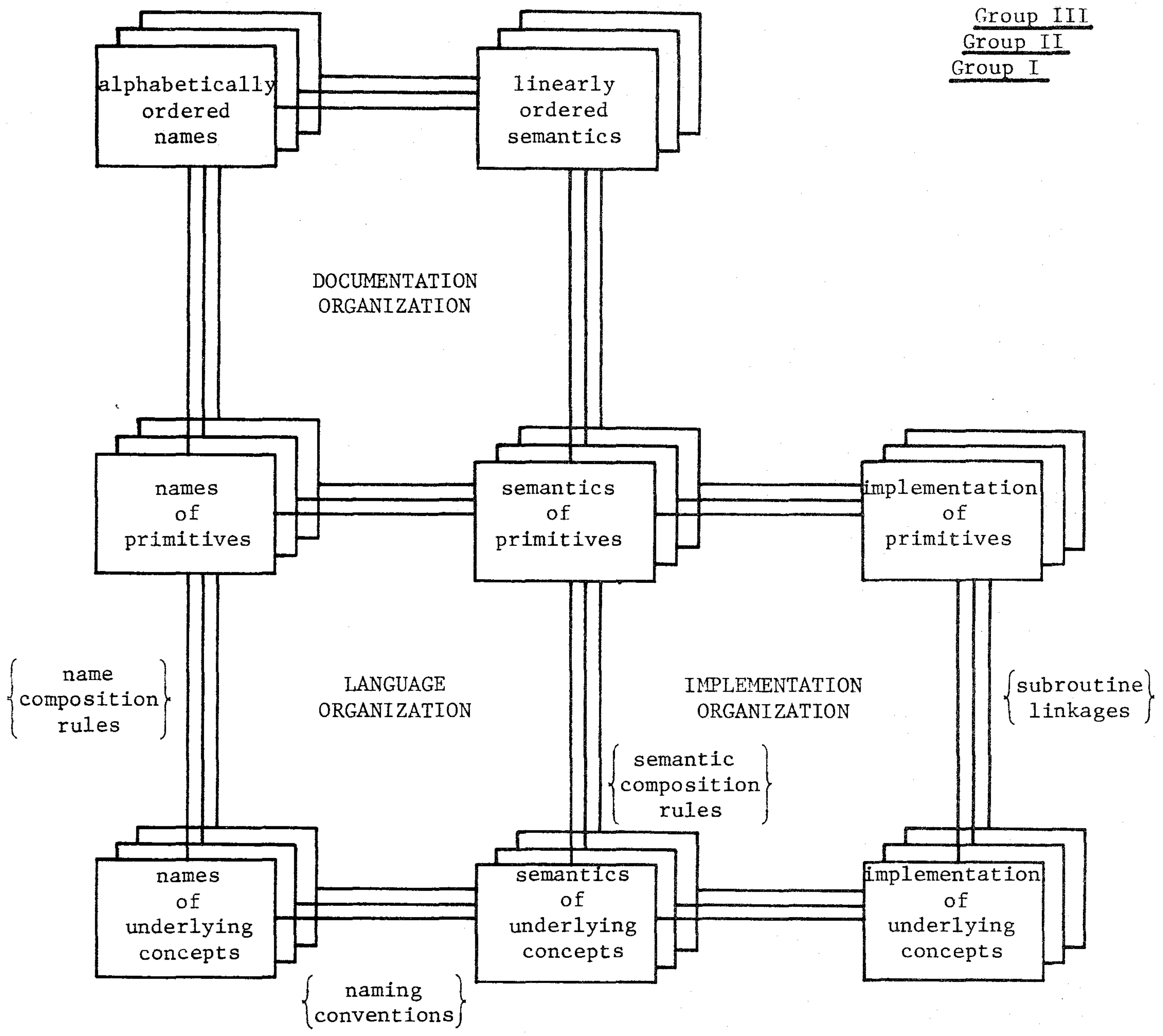

Group III
Group II
Group I
alphabetically ordered names
linearly ordered semantics
DOCUMENTATION ORGANIZATION
names of primitives
semantics of primitives
implementation of primitives
name composition rules
LANGUAGE ORGANIZATION
IMPLEMENTATION ORGANIZATION
subroutine linkages
semantic composition rules
names of underlying concepts
semantics of underlying concepts
implementation of underlying concepts
naming conventions

FIGURE 3 - Organizational Structure of GRASPER 1.0

This makes GRASPER easier to learn and retain, provides an indexing system for GRASPER documentation, and serves as an outline for well-structured implementations.

III. LANGUAGE DESIGN

III.1 Syntax

The syntax of GRASPER is largely that of its host. The syntax of atomic elements, lists, and operator calls depends exclusively on the host language. The syntactic design decisions embodied in GRASPER relate to operator names and the order of their arguments.

The name of each GRASPER operator systematically reflects the underlying concepts from which it is formed. For example, DOG is the GRASPER operator that destroys the outpointing edges of a node. Operators with similar names have similar semantics; operators with dissimilar names have dissimilar semantics. A novice quickly learns to predict the names of operators given their semantics, and the semantics of operators given their names. The alphabetical ordering of operators by name corresponds to a reasonable semantic ordering. This is particularly important to the organization of a reference manual where a linear ordering is required. All operator names are pronounceable to facilitate verbal interchanges.

The order of GRASPER operator arguments is consistent across all operators. Arguments which play similar roles are in similar positions. No operator has more than one optional argument, which always refers to a space and always appears in the last position.

III.2 Semantics

Graphs, by their nature, are pictorial. GRASPER primitives model what a person can easily do while looking at a drawing of a graph. The pictorial semantics of graphs motivates the semantics of GRASPER, while set theory and graph theory provide its formal foundation.

Most GRASPER operators have an optional space argument. When this is included, the scope of the operator is restricted to that space. When this argument is not included, a space named UNIVERSE is assumed. UNIVERSE is a special space maintained by GRASPER that contains all the nodes and edges in the GRASPER-GRAPH. If a node or edge is deleted from UNIVERSE, it is deleted from all spaces. For example, if an operator that destroys a node is given a space other than UNIVERSE, the node is only removed from that space. If that same operator is called with UNIVERSE or no space argument, it is removed from all spaces.

The Group I primitives form the basis of GRASPER. A small set of Group I concepts combine to form all of the Group I operators. These underlying concepts are divided into four categories, operator types, operator objects, object qualifiers, and qualifying objects. Every Group I operator has an operator type, an operator object, and an object qualifier, but not all have a qualifying object.

There are six Group I *operator types*. These types are grouped into two classes, functions and pseudo-functions. Functions are executed for the value they return. Pseudo-functions are executed for their effect. The function operator types include those that return sets of graph entities, those that return values of graph entities, and those that determine if graph entities exist. The pseudo-function operator types include those that create graph entities, those that destroy graph entities, and those that bind values to graph entities.

Operator objects of Group I operators specify the types of graph entities the operators manipulate. Operator objects include nodes, edges, spaces, and pairs. A *pair* is an edge and a node to which that edge is connected.

Object qualifiers include outpointing, inpointing, adjacent, and unqualified. Outpointing, inpointing, and adjacent qualifiers specify the means of accessing the operator object(s) from a node. *Outpointing* means to follow only edges which point away from a node; *inpointing*, only edges that point towards a node; and *adjacent*, all edges connected to a node. *Unqualified* is used when access is immediate through the object's name.

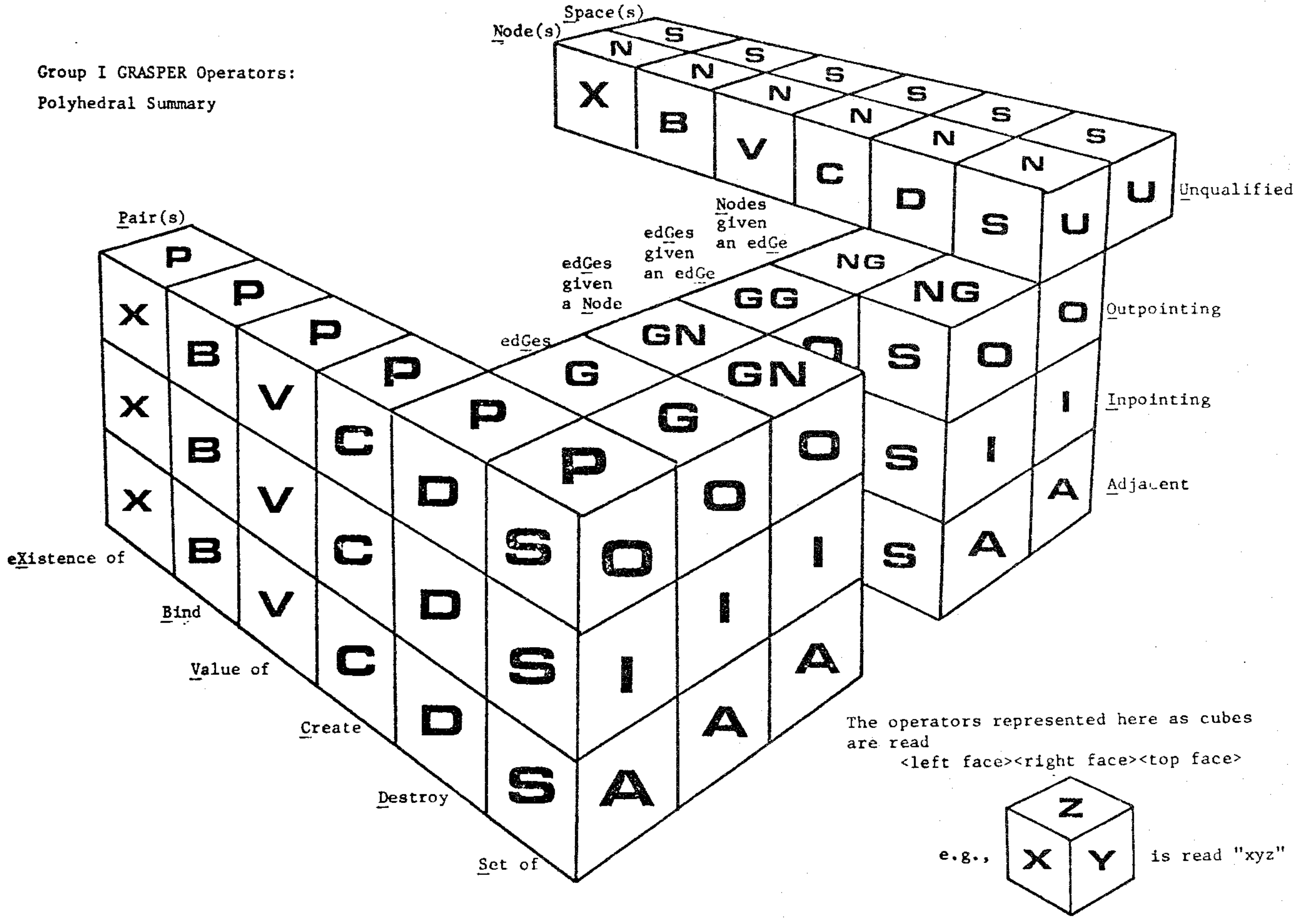
Qualifying objects are used to further restrict operator objects by specifying additional graph entities associated with them. Nodes and edges can be used as qualifying objects.

The Group I rules of operator composition describe how these underlying concepts can be combined to form Group I operators. The name of each operator is formed by concatenating single-letter abbreviations for the underlying concepts embodied in the operators. The role played by each letter is determined by its position in the operator name. These letter positions are ordered: operator type, object qualifier, operator object, and an optional qualifying object. For example, SAN returns the set of adjacent nodes of a node, DOG destroys the outpointing edges of a node, DOGN destroys the outpointing edges of a node that lead to a specified node, XIP tests whether an inpointing pair of a node exists, and CUN creates a node. The full set of Group I operators is summarized in Figure 4.

The Group II primitives are similar to the Group I primitives except they have greater scope. Where Group I primitives deal with units of GRASPER-GRAPHs, Group II primitives deal with major portions of GRASPER-GRAPHs. For example, the Group I operator CUN can only create a node in a single space, the Group II operator CREATE-NODE can create a node in multiple spaces with connecting edges and values.

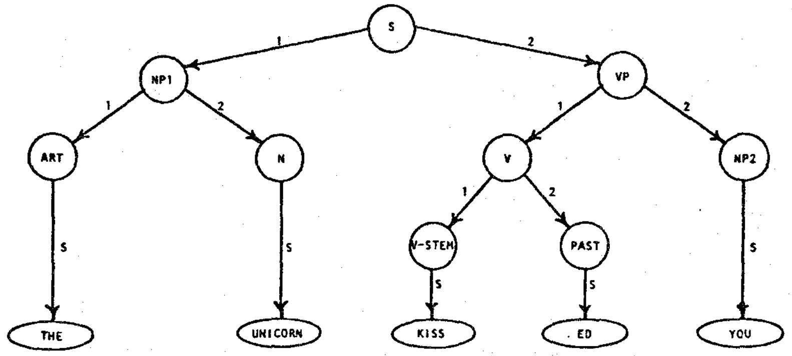
Group II DESCRIPTORs are specialized list structures that describe portions of graphs (see Figure 5). Most Group II operators either are passed a DESCRIPTOR as an argument or generate one to be returned.

Group I GRASPER Operators:
Polyhedral Summary

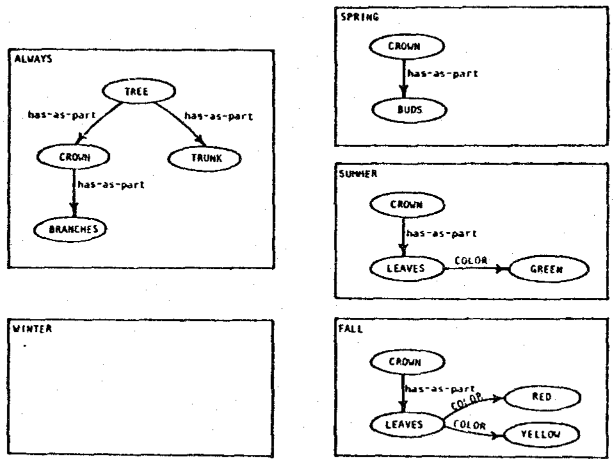


The operators represented here as cubes are read <left face><right face><top face>

FIGURE 4 - Group I Operator Summary



```
(NIL
 (S NIL ((1 NP1) (2 VP)))
 (NP1 NIL ((1 ART) (2 N)))
 (ART NIL ((S THE)))
 (N NIL ((S UNICORN)))
 (VP NIL ((1 V) (2 NP2)))
 (V NIL ((1 V-STEM) (2 PAST)))
 (V-STEM NIL ((S KISS)))
 (PAST NIL ((S ED)))
 (NP2 NIL ((1 PRO)))
 (PRO NIL ((S YOU)))
 (THE)
 (UNICORN)
 (KISS)
 (ED)
 (YOU))
```



```
((ALWAYS FALL SPRING SUMMER WINTER)
 (BRANCHES (ALWAYS))
 (BUDS (SPRING))
 (CROWN
  (ALWAYS FALL SPRING SUMMER)
  ((HAS-AS-PART BRANCHES (ALWAYS))
   (HAS-AS-PART BUDS (SPRING))
   (HAS-AS-PART LEAVES (FALL SUMMER))))
 (GREEN (SUMMER))
 (LEAVES
  (FALL SUMMER)
  ((COLOR GREEN (SUMMER))
   (COLOR RED (FALL))
   (COLOR YELLOW (FALL))))
 (RED (FALL))
 (TREE (ALWAYS)
       ((HAS-AS-PART CROWN (ALWAYS))
        (HAS-AS-PART TRUNK (ALWAYS))))
 (TRUNK (ALWAYS))
 (YELLOW (FALL)))
```

FIGURE 5 - Example Group II DESCRIPTORs

Group II operators are composed from a small set of Group II concepts, similar to the way that Group I operators are composed. Each is composed of an operator type and an operator object. Operator types include functions that return DESCRIPTORs describing portions of graphs and pseudo-functions that create portions of graphs from DESCRIPTORs, destroy portions of graphs, and pretty-print descriptions of portions of graphs. Operator objects can be nodes, spaces, or graphs.

The Group II rules of operator composition describe how operator types and objects can be combined. The name of each Group II operator is a hyphenated compound word consisting of an operator type followed by an operator object. For example, CREATE-GRAPH creates an entire graph from a DESCRIPTOR, DESCRIBE-SPACE returns a DESCRIPTOR of a space, and PRINT-NODE pretty-prints a description of a node. The Group II operators are summarized in Figure 6.

Group III primitives provide a means for specifying views through sets of spaces, control the GRASPER virtual memory system for GRASPER-GRAPH storage, and move GRASPER-GRAPHs in and out of long-term storage. Although the Group III operators can be viewed as being composed of operator types and operator objects, composition rules are not given since there are few shared underlying components.

A virtual space is a view through a set of spaces. GRASPER operators can be given virtual spaces as space arguments. The result

Group II GRASPER Operators: Polygonal Summary

DESTROY	DESTROY-GRAPH		
CREATE	CREATE-GRAPH	CREATE-NODE	
DESCRIBE	DESCRIBE-GRAPH	DESCRIBE-NODE	DESCRIBE-SPACE
PRINT	PRINT-GRAPH	PRINT-NODE	PRINT-SPACE
	GRAPH	NODE	SPACE

FIGURE 6 - Group II Operator Summary

is the same as if a space containing all the entities in the spaces that the virtual space is defined over were given to the operator. For example, the nodes BRANCHES, CROWN, GREEN, LEAVES, TREE, and TRUNK would be returned by the Group I operator SUN when given a virtual space defined over spaces ALWAYS and SUMMER in the graph of Figure 7. Virtual spaces can be used like CONTEXTs in other languages [RUL72,McD72] to represent incrementally different alternative worlds. The Group III function VIRTUAL-SPACE defines a virtual space.

GRASPER supports a virtual memory system for GRASPER-GRAPH storage. GRASPER-GRAPHs that are too large to be stored in primary memory are partitioned into pages and moved between primary and secondary memory as required. The Group III operators SET-SIZE and SIZE set and return the values of memory management parameters. The Group III operators REALIZE-UNIVERSE and VIRTUALIZE-UNIVERSE select between two different storage schemes for the space UNIVERSE. These issues are discussed further in the section on implementation design (pp. 36-38).

GRASPER-GRAPHs are moved between short-term and long-term storage by the Group III operators INPUT-GRAPH and OUTPUT-GRAPH.

GRASPER errors occur whenever requests are made through GRASPER operators that do not make sense in the context of the current GRASPER-GRAPH. These are usually references to nonexistent

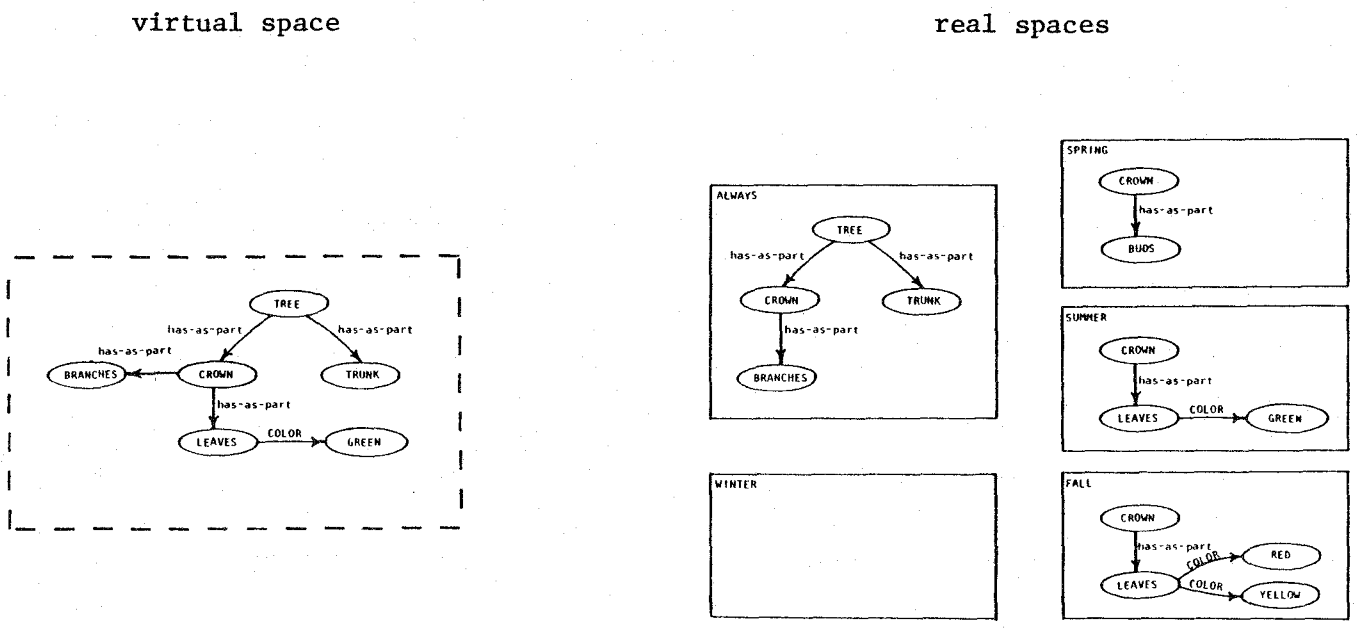


FIGURE 7 - Example virtual space defined over real spaces ALWAYS and SUMMER

GRASPER-GRAPH entities. When an error is detected an error message is printed. GRASPER error messages are short, concise, and understandable. They are in English, contain no criptic codes, and share the same basic format. Each error message includes the name of the operator causing the error, a description of the cause, and any offending arguments to the operator.

IV. DOCUMENTATION DESIGN

The user documentation for GRASPER eventually should consist of a reference manual and a primer. A reference manual and primer can not be successfully combined since they are inherently incompatible. A reference manual is like a dictionary. It should contain complete, concise descriptions of all primitives organized in a manner that allows the description of any primitive to be located quickly. A primer needs to introduce concepts in an incremental fashion to facilitate learning. This usually requires that oversimplified descriptions precede complete descriptions. A primer is meant to be read in its entirety. A reference manual, like a dictionary, is not meant to be read in its entirety.

IV.1 Primer

A GRASPER primer has not been written. Time permitted the writing of a primer or reference manual, but not both. Since a reference manual is of more general use over a longer period of time, it was written. Although a primer would present a better introduction to GRASPER, an experienced programmer can learn the language directly from the reference manual.

IV.2 Reference Manual

The "GRASPER 1.0 Reference Manual" [LOW78] contains a full description of each primitive in the language. Each description consists of the primitive's name, an informal definition, a formal definition, and numerous illustrations. (The reader is encouraged to refer to Figures 8-15 containing excerpts from the "GRASPER 1.0 Reference Manual" while reading this section.)

Most GRASPER primitives have acronyms as names. When this is the case, the derivation of the acronym is described. A phonetic spelling of the name is included whenever there is some doubt concerning its proper pronunciation.

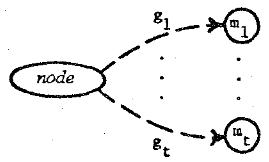
Informal definitions consist of prose descriptions of the primitives including all error conditions. Pictorial descriptions accompany these whenever appropriate. This is in keeping with the pictorial motivation of the language.

Formal definitions, written in set notation, are included to assist both users and implementers. Error conditions are an integral part of these formal definitions. Unlike the formal definitions of many other languages, most of these are short and easy to understand. Users and implementers alike are encouraged to use these since they provide the most accurate and concise description of GRASPER.

(DOG *node*)[1] Destroy Outpointing edGes \'dȯg\

Informal Definition

The pseudo-function DOG is an EXPR which has the effect of destroying all outpointing edges of *node*. Given *node*, DOG destroys all edges g_i where for each i, edge g_i points to some node m_i from *node*. If *node* has no such edges, DOG has no effect. DOG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DOG[n] = n

with effects:

NGN := NGN − $\{(n\ g\ m) \mid g \in G,\ m \in N\}$

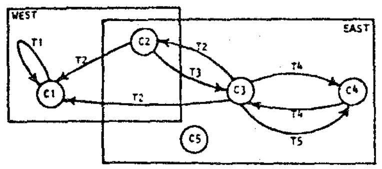
NGNSV := NGNSV − $\{(((n\ g\ m)\ s)\ v) \mid g \in G,\ m \in N,\ s \in S,\ v \in V\}$

error condition:

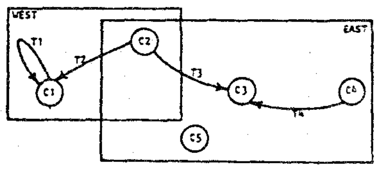
− $n \notin N$

[1]See alternative form on page 116.

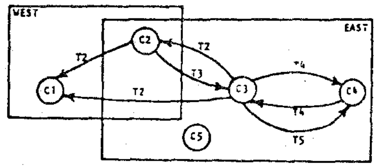
Illustrations



```
?(DOG 'C3)
C3
```



```
?(DOG 'C1)
C1
```



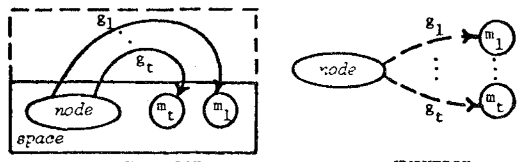
```
?(DOG 'C5)
C5

?(DOG 'CX)
*** DOG ERROR: CX IS NOT A NODE
```

FIGURE 8 - Excerpt from "GRASPER 1.0 Reference Manual"

(DOG *node space*)[1] Destroy Outpointing edGes \'dóg\

Informal Definition

The pseudo-function DOG is an EXPR which has the effect of destroying all outpointing edges of *node* in *space*. Given *node* and *space*, DOG removes all edges g_i from *space* where for each i, edge g_i points to some node m_i from *node*. If *space* is UNIVERSE, DOG removes each such edge g_i from all spaces. If *node* has no such edges, DOG has no effect. DOG returns *node*.



space ≠ UNIVERSE *space* = UNIVERSE

error conditions:

- *node* does not exist in *space*
- *space* does not exist

Formal Definition

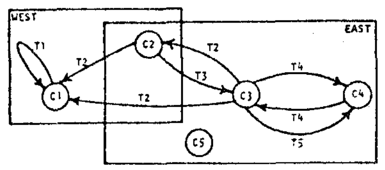
```
DOG[n,s] = n
   with effects:
      if s = UNIVERSE
         then DOG[n]
         else NGNSV := NGNSV - {(((n g m) s) v)|g ∈ G, m ∈ N, v ∈ V}
```

error conditions:

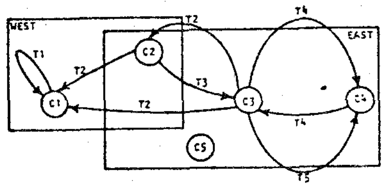
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 114.

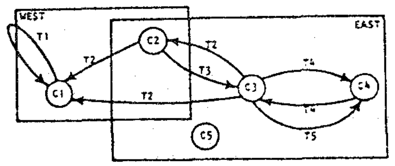
Illustrations



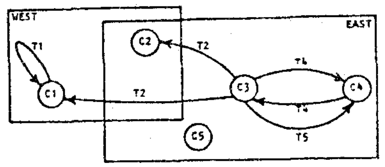
```
?(DOG 'C3 'EAST)
C3
```



```
?(DOG 'C1 'WEST)
C1
```



```
?(DOG 'C2 'UNIVERSE)
C2
```



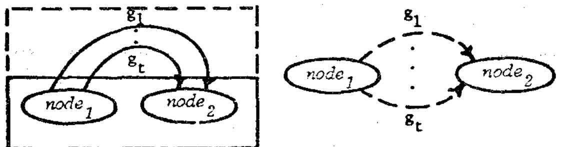
```
?(DOG 'C5 'EAST)
C5
?(DOG 'CX 'EAST)
*** DOG ERROR: CX IS NOT A NODE IN SPACE EAST
?(DOG 'C3 'SX)
*** DOG ERROR: SX IS NOT A SPACE
```

FIGURE 9 - Excerpt from "GRASPER 1.0 Reference Manual"

(DOGN $node_1$ $node_2$ *space*)[1] Destroy Outpointing edGes given a Node \'dôg-in\

Informal Definition

The pseudo-function DOGN is an EXPR which has the effect of destroying all outpointing edges of $node_1$ that point to $node_2$ in *space*. Given $node_1$ and $node_2$, DOGN removes all edges g_i from *space* where for each i, g_i points from $node_1$ to $node_2$. If *space* is UNIVERSE, DOGN removes each such edge g_i from all spaces. If no such edges exist, DOGN has no effect. DOGN returns $node_1$.



space ≠ UNIVERSE *space* = UNIVERSE

error conditions:
- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- *space* does not exist

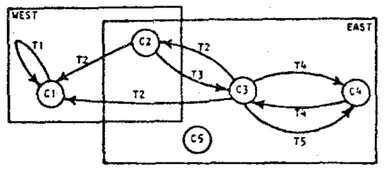
Formal Definition

```
DOGN[n,m,s] = n
  with effects:
    if s = UNIVERSE
      then DOGN[n,m]
      else NGNSV := NGNSV - {(((n g m) s) v)|g ∈ G, v ∈ V}
```

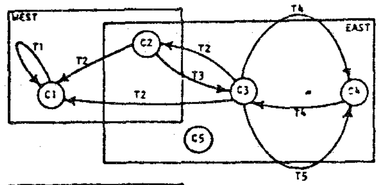
error conditions:
- ((n s) v) ∉ NSV for all v ∈ V
- ((m s) v) ∉ NSV for all v ∈ V
- s ∉ S

[1]See alternative form on page 122.

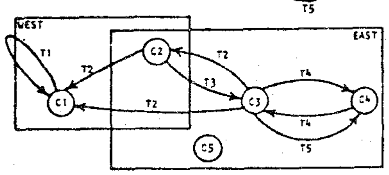
Illustrations



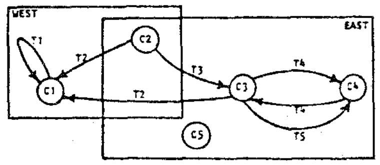
```
?(DOGN 'C3 'C4 'EAST)
C3
```



```
?(DOGN 'C1 'C1 'WEST)
C1
```



```
?(DOGN 'C3 'C2 'UNIVERSE)
C3
```



```
?(DOGN 'C5 'C4 'EAST)
C5

?(DOGN 'CX 'C4 'EAST)
*** DOGN ERROR: CX IS NOT A NODE IN SPACE EAST

?(DOGN 'C3 'CX 'EAST)
*** DOGN ERROR: CX IS NOT A NODE IN SPACE EAST

?(DOGN 'C3 'C4 'SX)
*** DOGN ERROR: SX IS NOT A SPACE
```

FIGURE 10 - Excerpt from "GRASPER 1.0 Reference Manual"

A generous number of illustrations accompanies the description of each primitive. These illustrate both appropriate and inappropriate uses of the primitives. Drawings are included whenever useful.

The information describing each GRASPER primitive is localized to a few consecutive pages in the manual. This helps assure that a reader will not overlook pertinent information. The information on facing pages almost always pertains to the same primitive. This visually separates descriptions of different primitives. Alternative forms of a single primitive are described as if they are distinct primitives, but are always described on consecutive pages. Footnotes direct the reader between descriptions of alternative forms.

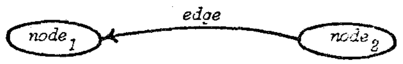
The typography of the manual is based on the premise that things which are the same should be visually similar, and things which are different should be visually distinct. Consistent formatting conventions are used throughout the manual. This helps a user to find desired information by sight rather than by reading. Different headings, margin settings, type fonts, page positions, and spacing are some of the techniques used to produce a desirable visual impact. A reader need not be aware of the typographic conventions to benefit from them [SAC79].

The manual's indexing system allows the description of any primitive to be located quickly. Each group of GRASPER primitives is

(XIP *node*$_1$ *edge* *node*$_2$)[1] eXistence of Inpointing Pair \'zip\

<u>Informal Definition</u>

The function XIP is an EXPR which tests for the existence of the inpointing pair (*edge* *node*$_2$) of *node*$_1$. Given *node*$_1$, *edge*, and *node*$_2$, XIP returns T if *edge* points from *node*$_2$ to *node*$_1$ and NIL if it does not.



error conditions:

- *node*$_1$ does not exist
- *node*$_2$ does not exist

<u>Formal Definition</u>

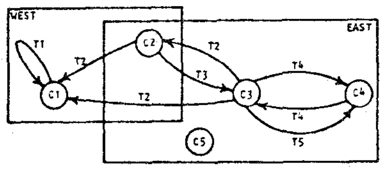
```
XIP[n,g,m] = x
   where if (m g n) ∈ NGN
            then x = T
            else x = NIL
```

error conditions:

- $n \notin N$
- $m \notin N$

[1]See alternative form on page 236.

<u>Illustrations</u>



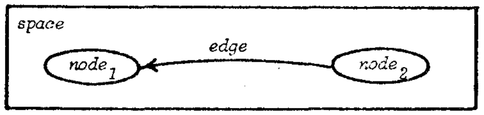
```
?(XIP 'C3 'T4 'C4)
T

?(XIP 'C4 'T4 'C3)
T

?(XIP 'C1 'T2 'C3)
T

?(XIP 'C1 'T1 'C1)
T

?(XIP 'C2 'T3 'C3)
NIL

?(XIP 'C2 'TX 'C3)
NIL

?(XIP 'CX 'T4 'C4)
*** XIP ERROR: CX IS NOT A NODE

?(XIP 'C3 'T4 'CX)
*** XIP ERROR: CX IS NOT A NODE
```

FIGURE 11 - Excerpt from "GRASPER 1.0 Reference Manual"

(XIP *node*$_1$ *edge* *node*$_2$ *space*)[1] eXistence of Inpointing Pair \'zip\

Informal Definition

The function XIP is an EXPR which tests for the existence of the inpointing pair (*edge* *node*$_2$) of *node*$_1$ in *space*. Given *node*$_1$, *edge*, *node*$_2$, and *space*, XIP returns T if *edge* points from *node*$_2$ to *node*$_1$ in *space*, and NIL if it does not.



error conditions:

- *node*$_1$ does not exist in *space*
- *node*$_2$ does not exist in *space*
- *space* does not exist

Formal Definition

$\mathrm{XIP}[n,g,m,s] = x$

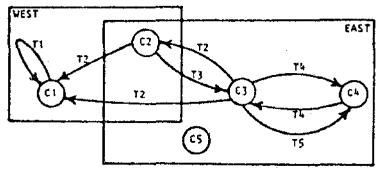
where if $(((m\ g\ n)\ s)\ v') \in \mathrm{NGNSV}$ for some $v' \in V$
then $x = T$
else $x = \mathrm{NIL}$

error conditions:

- $((n\ s)\ v) \notin \mathrm{NSV}$ for all $v \in V$
- $((m\ s)\ v) \notin \mathrm{NSV}$ for all $v \in V$
- $s \notin S$

[1] See alternative form on page 234.

Illustrations



```
?(XIP 'C3 'T4 'C4 'EAST)
T

?(XIP 'C1 'T2 'C2 'WEST)
T

?(XIP 'C1 'T2 'C3 'UNIVERSE)
T

?(XIP 'C3 'T2 'C2 'EAST)
NIL

?(XIP 'C3 'TX 'C4 'EAST)
NIL

?(XIP 'CX 'T4 'C4 'EAST)
*** XIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(XIP 'C3 'T4 'CX 'EAST)
*** XIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(XIP 'C3 'T4 'C4 'SX)
*** XIP ERROR: SX IS NOT A SPACE
```

FIGURE 12 - Excerpt from "GRASPER 1.0 Reference Manual"

(DESCRIBE-NODE *node*)[1]

<u>Informal Definition</u>

The function DESCRIBE-NODE is an EXPR which returns a NODE-DESCRIPTOR for *node* in the existing GRAPH. Given *node*, DESCRIBE-NODE returns a NODE-DESCRIPTOR describing *node* including information about all the spaces *node* is in. DESCRIBE-NODE does not describe NIL values, inclusion in the universal space when the universal value is NIL, or entities whose corresponding switches are off.

error condition:

- *node* does not exist

<u>Formal Definition</u>

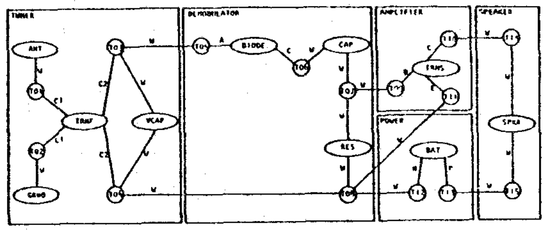
DESCRIBE-NODE[n] = DESCRIBE-NODE[n, sus[n] ∪ {UNIVERSE}]

error condition:

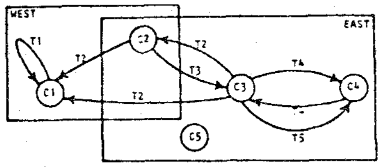
- $n \notin N$

[1]See alternative form on page 284.

<u>Illustrations</u>

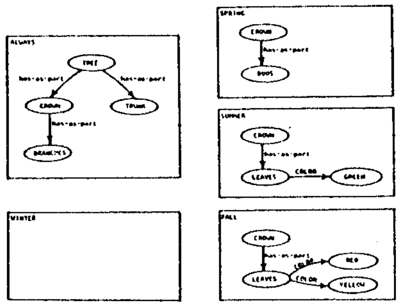


```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'T04))

SWITCH-S = T                                          SWITCH-SV = T
SWITCH-N = T                   SWITCH-NS = T          SWITCH-NV = T
SWITCH-OP = T                  SWITCH-OPS = T         SWITCH-OPV = T
SWITCH-IP = T                  SWITCH-IPS = T         SWITCH-IPV = T

(T04 (TUNER)
     ((C2 TRNF (TUNER)) (W T08) (W VCAP (TUNER)))
     ((C2 TRNF (TUNER)) (W T08) (W VCAP (TUNER))))
```

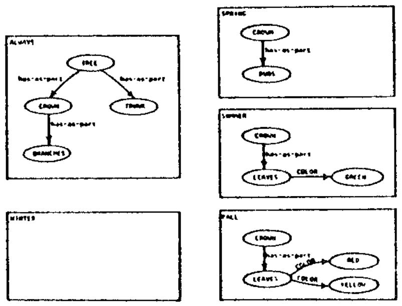


```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'C2))

SWITCH-S = T                                          SWITCH-SV = T
SWITCH-N = T                   SWITCH-NS = T          SWITCH-NV = T
SWITCH-OP = T                  SWITCH-OPS = T         SWITCH-OPV = T
SWITCH-IP = T                  SWITCH-IPS = T         SWITCH-IPV = T

(C2 (EAST UNIVERSE = (100 110) WEST)
    ((T2 C1 (UNIVERSE = 70 WEST)) (T3 C3 (EAST UNIVERSE = 80)))
    ((T2 C3 (EAST UNIVERSE = 70))))
```

FIGURE 13 - Excerpt from "GRASPER 1.0 Reference Manual"



```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'CROWN))

SWITCH-S = T                                       SWITCH-SV = T
SWITCH-N = T              SWITCH-NS = T            SWITCH-NV = T
SWITCH-OP = T             SWITCH-OPS = T           SWITCH-OPV = T
SWITCH-IP = T             SWITCH-IPS = T           SWITCH-IPV = T

(CROWN
 (ALWAYS FALL SPRING SUMMER)
 ((HAS-AS-PART BRANCHES (ALWAYS))
  (HAS-AS-PART BUDS (SPRING))
  (HAS-AS-PART LEAVES (FALL SUMMER)))
 ((HAS-AS-PART TREE (ALWAYS))))
```



```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'CROWN))

SWITCH-S = T                                       SWITCH-SV = T
SWITCH-N = T              SWITCH-NS = NIL          SWITCH-NV = T
SWITCH-OP = T             SWITCH-OPS = NIL         SWITCH-OPV = T
SWITCH-IP = T             SWITCH-IPS = NIL         SWITCH-IPV = T

(CROWN NIL
       ((HAS-AS-PART BRANCHES) (HAS-AS-PART BUDS)
                               (HAS-AS-PART LEAVES))
       ((HAS-AS-PART TREE)))
```

```
?(DESCRIBE-NODE 'NX)
*** DESCRIBE-NODE ERROR: NX IS NOT A NODE
```

FIGURE 14 - Excerpt from "GRASPER 1.0 Reference Manual"

described in a separate section of the manual. The primitives described within each section are in alphabetical order. Running headings and footings contain all the information a user is likely to need to navigate through the manual. The section names appear at the tops of the pages and the names of the primitives being described appear in the lower outside corners. A user locates a description of a primitive by first flipping through the manual until the name of the desired group appears at the tops of the pages and then leafing through that section, alphabetically directed by the names of the primitives in the lower corners, until the desired name appears.

Even when the name of a primitive is unknown, often it is possible to find a primitive with the desired effect. This results since physical locality in the manual corresponds to semantic locality. Primitives that are alphabetically near are physically near in the manual and semantically similar. A user can find a portion of the manual describing semantically similar primitives and then scan that area for a primitive with the desired effect.

If this indexing system fails, the user still has one last recourse. The manual has a combination index and glossary. This directs the user by page numbers to explanations of key GRASPER concepts. It also defines many of the terms and notations used in the manual.

(OUTPUT-GRAPH *file*)

Informal Definition

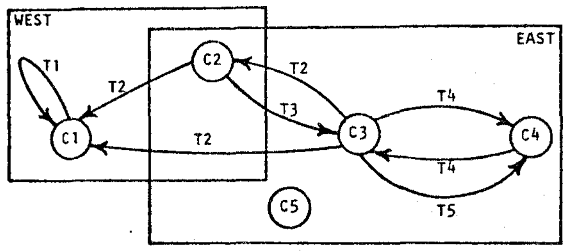
The pseudo-function OUTPUT-GRAPH is an EXPR which has the effect of outputing the current GRAPH to *file*.[1] If *file* does not already exist, OUTPUT-GRAPH creates it and stores GRAPH in it. If *file* does exist, its contents are replaced with GRAPH. OUTPUT-GRAPH returns *file*.

Formal Definition

OUTPUT-GRAPH[f] = f

[1]The means of specifying a file is implementation dependent.

Illustrations



```
 ?(OUTPUT-GRAPH 'FILE1)
FILE1

 ?(PROGN (OUTPUT-GRAPH 'FILE1)
         (DESTROY-GRAPH)
         (CUN 'C10)
         (INPUT-GRAPH 'FILE1))
FILE1
```

FIGURE 15 - Excerpt from "GRASPER 1.0 Reference Manual"

A separate section of the manual describes each error message. These also are in alphabetical order. Each error message description includes a brief description of the error condition that causes it to be printed and a list of the GRASPER operators that can generate that error condition.

V. IMPLEMENTATION DESIGN

GRASPER has been implemented with LISP 1.5 [McC65], extended to include random access I/O capabilities, serving as the host language. This implementation is faithful to the definition of GRASPER in the "GRASPER 1.0 Reference Manual." Implementation design decisions can be grouped into two classes: those independent of the host language and those specifically related to the choice of LISP 1.5 as the host.

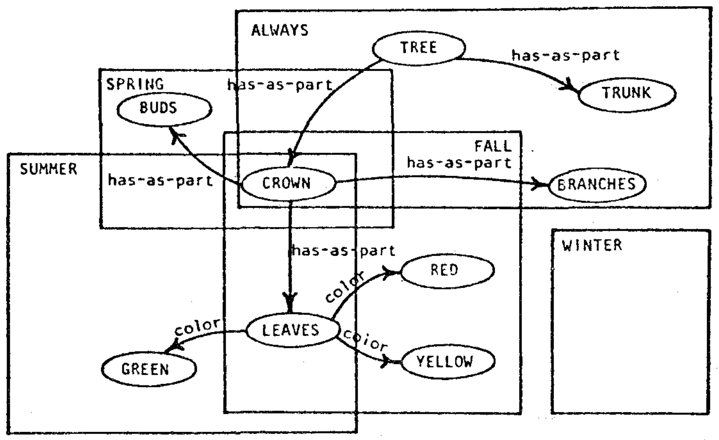
V.1 Host-Independent Design Decisions

GRASPER was implemented with human performance as a guide. The speed of each operation is related to the difficulty a person would have performing that same operation on a drawing of a graph. For example, since a person can follow inpointing and outpointing edges with equal ease, the implementation stores each edge twice, once as an inpointing edge and again as an outpointing edge. This makes operations involving inpointing edges as efficient as those involving outpointing edges. This technique gives users a guide to the relative efficiency of GRASPER operations.

GRASPER-GRAPHs are implemented as collections of space descriptions. Each _space description_ consists of a space's value and a node description for each node in that space. A _node description_ contains all the information about a node in a space: its value, its connecting edges, and their values. A description of each real (non-virtual) space is maintained in memory. The description of a virtual space is not stored but dynamically determined from the spaces it is defined over. Virtual spaces are more memory efficient but have slower access times than real spaces.

The universal space is normally maintained as a real space. In this _real-UNIVERSE mode_, UNIVERSE contains a complete inversion by nodes of all the information contained in the other spaces. However, GRASPER can be instructed to maintain UNIVERSE as a virtual space defined over all other spaces. When GRASPER is in _virtual-UNIVERSE mode_, the amount of memory required to store the graph is approximately one-half that of the same graph in real-UNIVERSE mode. Updates to the graph are faster in virtual-UNIVERSE mode since UNIVERSE need not be updated each time a space is updated. But as with all virtual spaces, retrieval times for UNIVERSE are slower in virtual-UNIVERSE mode. Several Group III operators control the mode of UNIVERSE.

These modes correspond to two different ways of drawing graphs (see Figure 16). One way is to draw each node once, overlapping spaces which contain common nodes. Another way is to draw each space separately, duplicating nodes which exist in more than one space. Drawn the first way (corresponding to real-UNIVERSE mode) all the information about a node is gathered in a single place. Drawn the real-UNIVERSE mode



virtual-UNIVERSE mode

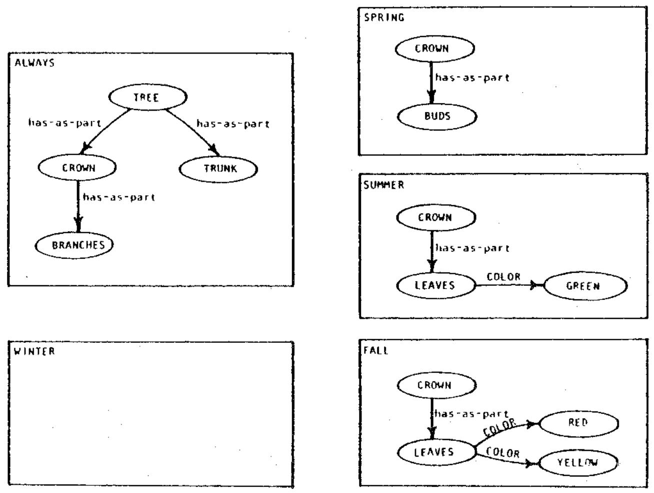


FIGURE 16 - Alternative drawings of an example GRASPER-GRAPH

second way (corresponding to virtual-UNIVERSE mode) the information about a node is scattered among all the spaces in which that node exists.

The implementation was greatly influenced by the requirement that it support large graphs. Since there is no guarantee that large graphs will fit in primary memory, the implementation supports a software-level virtual memory management system for GRASPER-GRAPH storage. Graphs that are too large to be stored in primary memory are partitioned into pages and stored in secondary memory. Pages are moved between primary and secondary memory as required. Spaces, meaningful subgraphs containing logically related information, are used like segments in other virtual memory management systems to partition graphs along meaningful lines. Users can reduce paging by grouping references to the same spaces in their programs. Previous attempts to reduce paging in graph data bases, without the aide of spaces, were largely unsuccessful [BIS77]. The memory management system requires a minimal amount of user interaction. Users without special memory management requirements do not need to make memory management declarations.

Many GRASPER operators are defined to return sets. In this implementation, (alphabetically) ordered sets are returned. This permits the writing of more efficient user programs; makes it easier for users to visually locate particular elements in returned results; and, since the same ordering is used by the memory management system for graph storage, paging is minimized when graph references are similarly ordered.

A high read/write ratio was assumed with respect to graphs. Therefore, some additional expense during write operations is incurred in favor of more efficient read operations.

The organization of the language and reference manual carries over to the implementation. Source files are structured in a way that closely resembles this organization. Semantically similar primitives are implemented along similar lines, with common underlying concepts manifest as common subroutines. A test system for the implementation generates the examples that appear in the manual. DESCRIPTORs for all the graphs in the manual are included in the implementation to assist users while learning the language.

V.2 LISP 1.5 Dependent Design Decisions

Node descriptions are implemented as multi-leveled linked lists. It was assumed that the number of edges attached to any given node would be small enough for linear retrieval. Space descriptions are implemented as balanced binary (AVL) trees [ADE62] of node descriptions. These trees are implemented as linked lists. Another AVL tree is used to store all space descriptions. Therefore, the graph data base is an AVL tree of AVL trees of multi-leveled lists. Figure 17 illustrates this structure.

If a space description becomes large, it is split into two partial space descriptions. If a partial space description becomes small, it is merged with another partial space description of that space.

Graphs that are too large to be stored in primary memory are stored in secondary memory on random access files. Complete (undivided) space descriptions and partial space descriptions, the pages of this virtual memory system, are loaded into primary memory as required. Deactivated pages are paged-out using a least-recently-used strategy. Page size and the amount of primary memory available for graph storage are controlled by the user through several Group III operators.

The results of GRASPER operators must be usable in the same way as the result of any other LISP function. A result must not "disappear" from the user when paging occurs. In this implementation, pages are transferred into primary memory by copying them into LISP free space. The normal garbage collection process "deletes" paged-out pages without deleting those portions still referenced by the user.

This approach was selected for several reasons. First, there was a strong desire for the implementation to be highly compatible with

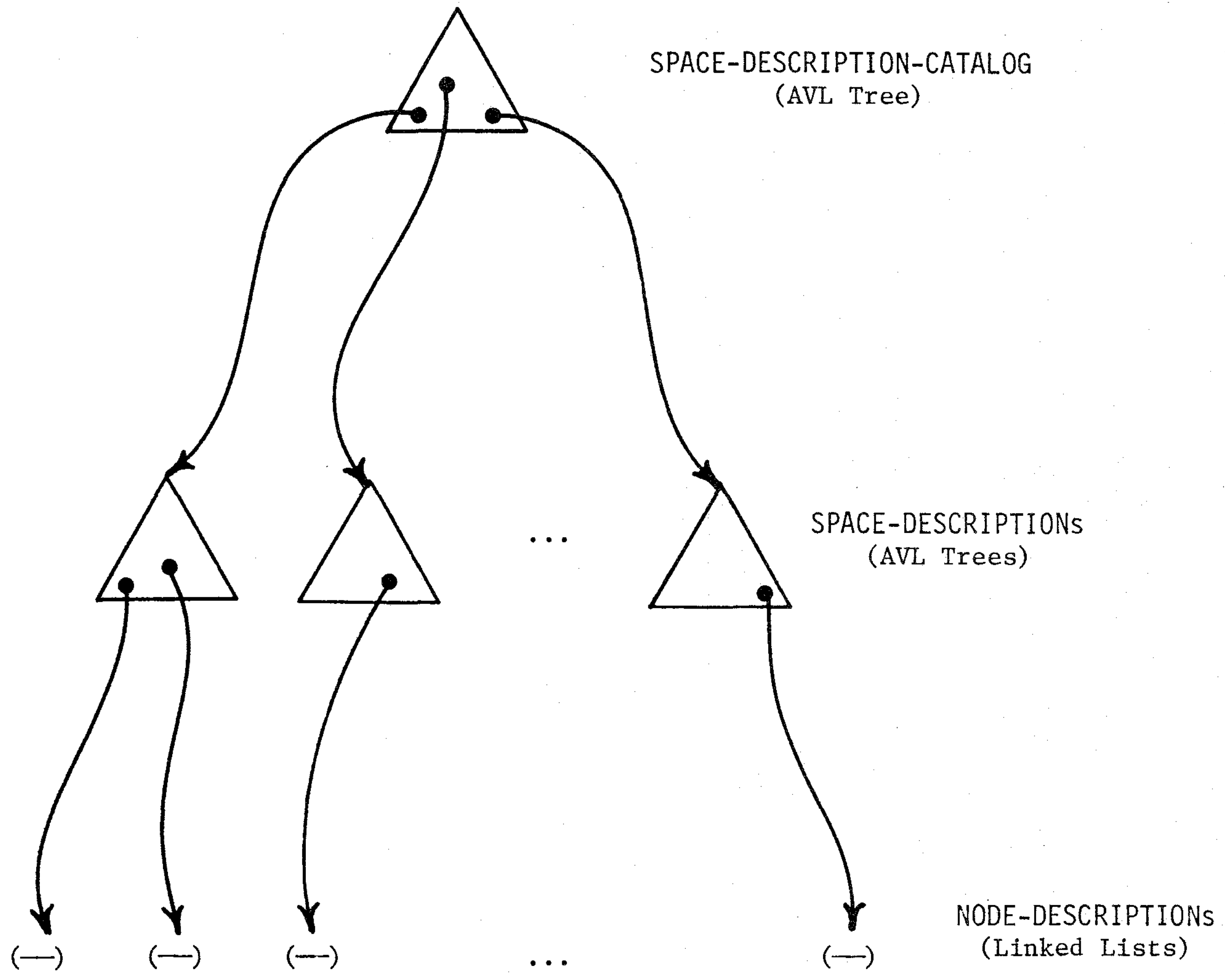


FIGURE 17 - GRASPER 1.0 Data Base: An AVL tree of AVL trees of linked lists

LISP. This approach allowed the memory manager to be implemented entirely at the LISP level. Second, this combined storage approach for graphs and LISP data permits better memory utilization since available memory is dynamically allocated. Third, since pages are implemented as LISP lists, page size variations do not fragment memory. Finally, this combined storage approach is nicely suited to LISP implementations which have concurrent garbage collection.

This implementation includes a set of auxillary operators that extend LISP 1.5, making it an improved host environment for GRASPER. These operators include: common set-theoretic operators, additional mapping operators, paging monitors, and operators which print the GRASPER news. The additional mapping operators, functional combinators [FRI74], provide a more general mapping facility than is available in LISP 1.5. Because GRASPER operators have multiple arguments, a more sophisticated mapping facility is desirable. Paging monitors aid users with extreme efficiency requirements in tailoring their programs for high memory management performance. The news facility informs users of the peculiarities of their particular implementation installation.

VI. POSSIBLE EXTENSIONS

GRASPER has been constantly evolving over the last three years. GRASPER 1.0 represents a stable set of concepts arrived at through this evolutionary process. A number of shortcomings still remain. This section describes a few ideas for the future development of GRASPER.

The greatest single improvement to GRASPER probably would be the addition of a graph matcher. This could take the form of Group II operators that perform analogous operations to the Group I "X" (existence-of) and "S" (set-of) operators. Such operators might use DESCRIPTORs containing variables to express graph patterns.

A set of Group II plotting operators for the graphic display of GRASPER-GRAPHs would be extremely useful. There should be a Group II PLOT operator for every PRINT operator. This would make the pictorial aspect of GRASPER more apparent.

Graph typing would be a welcome addition. This would allow a user to associate meta-descriptions with spaces. These meta-descriptions, possibly graphs themselves, would describe what graphical configurations can legally appear on their associated spaces. A type error would occur whenever a type is about to be violated.

Another addition might be user-defined automatically-activated procedures triggered on particular graph operations. These are commonly referred to as "demons" [HEW71]. Whenever the triggering condition of a demon occurs, control automatically is transferred to that demon.

Multiple storage modes have proven to be an extremely useful feature. Currently, virtual-UNIVERSE and real-UNIVERSE are the only storage modes available. A third storage mode, where only the universal view is maintained, also might prove useful.

GRASPER implementations might benefit from the addition of a software cache, retaining the most recently used space and node descriptions. If references to the same spaces and nodes tend to be grouped, such a cache would shorten the average access time for graph entities.

VII. SUMMARY

The primary feature of GRASPER's design is that the language, its documentation, and its implementation all share a common organizational structure. This structure lends itself equally well to the language, its documentation, and its implementation.

GRASPER primitives are systematically composed from a small set of underlying concepts. They are divided into three groups according to their scope. The name of each GRASPER primitive directly reflects the group it is in and the underlying concepts from which it is formed. The alphabetical ordering of primitives by name, within each group, corresponds to a reasonable semantic ordering.

This structure organizes a large set of primitives in a cognitively efficient way. A user quickly learns to predict the names of primitives given their semantics and the semantics of primitives given their names. Each group of GRASPER primitives is described in a separate section of the reference manual. The primitives described within each section are in alphabetical order. This allows the description of any primitive with a particular name or desired effect to be quickly located. Semantically similar primitives have similar implementations. Common underlying concepts are manifest as common subroutines.

This consistency in design helps to make GRASPER a more reliable system for users and implementers alike.

BIBLIOGRAPHY

[ADE62] G.M. Adel'son-Vel'skii and E.M. Landis (1962). An Algorithm for the Organization of Information. Soviet Math 3, No. 5, pp. 1259-1263.

[BIS77] Roberto Bisiani (1977). Paging Behavior of Knowledge Networks. Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[FRI77] Daniel P. Friedman (1977). Functional Combination. Computer Languages, Vol. 3, pp. 31-35.

[HAN78a] Allen R. Hanson and Edward M. Riseman (1978). Segmentation of Natural Scenes. Computer Vision Systems (A. Hanson and E. Riseman, Eds.), Academic Press, New York, New York.

[HAN78b] Allen R. Hanson and Edward M. Riseman (1978). VISIONS: A Computer System for Interpreting Scenes. Computer Vision Systems (A. Hanson and E. Riseman, Eds.), Academic Press, New York, New York.

[HEN75] Gary G. Hendrix (1975). Partitioned Networks for the Mathematical Modeling of Natural Language Semantics. Department of Computer Science Technical Report TR NL-28, University of Texas, Austin, Texas.

[HEW71] Carl Hewitt et al. (1971). Procedural Embedding of Knowledge in Planner. Proceedings of IJCAI2, London.

[LOW78] John D. Lowrance (1978). GRASPER 1.0 Reference Manual. COINS Technical Report 78-20, University of Massachusetts, Amherst, Massachusetts.

[McC65] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin (1965). LISP 1.5 Programmers Manual. The MIT Press, Massachusetts Institute of Technology, Cambridge, Massachusetts.

[PRA71] Terrance W. Pratt and Daniel P. Friedman (1971). A Language Extension for Graph Processing and Its Formal Semantics. CACM, Vol. 14, No. 7, pp. 460-467.

[RUL72] John F. Rulifson, Jan A. Derksen, and Richard J. Waldinger (1972). QA4: A Procedural Calculus for Intuitive Reasoning. SRI Technical Note 73, Menlo Park, California.

[SAC79] Jonathan Sachs (1979). Some Notes on Typography in Technical Documentation. Systems Documentation Newsletter, Vol. 5, No. 5, pp. 10-15.