GRASPER 1.0 Reference Manual

John D. Lowrance

COINS Technical Report 78-20

Computer and Information Science

University of Massachusetts at Amherst

Computers

Theory of Computation

Cybernetics

GRASPER 1.0 Reference Manual

John D. Lowrance

COINS Technical Report 78-20

(December 1978)

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

GRASPER 1.0 Reference Manual

BY

JOHN D. LOWRANCE

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

Second printing, March, 1980

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

PREFACE

This document constitutes a reference manual for GRASPER 1.0, a programming language extension that provides graph processing capabilities. This document is not designed to serve as an introductory text.

The information concerning each GRASPER 1.0 primitive -- including an informal definition, a formal definition, and illustrations -- has been localized to a few consecutive pages. A user can quickly locate this information for any GRASPER 1.0 primitive through the manual's indexing system. GRASPER 1.0 primitives are divided into a few major groups. Each group is described in a separate section of this manual. Section names appear at the top of the pages. The primitives described in each section are in alphabetical order. Their names appear in the lower corners. Thus, a user flips through the manual until the name of the desired group appears at the top of the pages and then leafs through the pages of that section, alphabetically directed by the names in the lower corners, until the desired primitive appears.

The formal definitions of GRASPER 1.0 primitives are included in this manual to assist both the GRASPER 1.0 user and implementer. Most of the formal definitions are short and easy to understand. The reader is encouraged to utilize these since they provide the most accurate and concise description of GRASPER 1.0.

This manual assumes some familiarity with elementary LISP 1.5 concepts. A reader unfamiliar with LISP 1.5 will find introductions in the following references [ALL78, FRI74, McC65, SIK75, WIN77, WIS67].

ACKNOWLEDGEMENTS

GRASPER 1.0 was developed as a data base support facility for the VISIONS system [HAN78a,b]. As the data base requirements of VISIONS changed over the past three years, an implementation of GRASPE 1.5 was gradually transformed into what is now GRASPER 1.0. During that time a great many people contributed to the design, documentation, and implementation of GRASPER 1.0. Without those contributions the results certainly would have been far less satisfactory. The following people were especially giving of their time and ideas.

-- Allen R. Hanson and Edward M. Riseman supervised all aspects of the project. Their comments motivated several of the central features of the language.

-- Henry F. Ledgard critically reviewed several different versions of the language. His comments prompted numerous modifications.

-- Janet E. Turnbull's exceptional clerical skills transformed a rough draft into a polished manual far exceeding any reasonable expectations.

-- Daniel D. Corkill made considerable contributions to the design and implementation of the GRASPER memory management system. Many of his suggestions concerning other aspects of the language influenced its design.

-- Richard S. Brooks, Daniel P. Friedman, Daryl T. Lawton, Thomas D. Williams, and Bryant W. York provided useful feedback on many aspects of the language.

-- Kurt Konolige, Bill Torcaso, and Richard L. Hudson provided ALISP support for the development of GRASPER. This support included some non-trivial modifications and extensions of ALISP.

-- Robert P. Heller optimized and compiled the ALISP implementation. He also modified this implementation to run using LISP F3 as the host language.

My special thanks go to all of these people for their time and effort.

Finally, I want to thank those early users of GRASPER who suffered through the changing implementation and incomplete manuals. I hope the results suit their needs.

John D. Lowrance

TABLE OF CONTENTS

Page

INTRODUCTION

Graphs, diagrams consisting of points connected by lines or arrows, are commonly used to depict situations of interest. Sociograms (psychology), simplexes (topology), circuit diagrams (physics, engineering), organization structures (economics), state transition diagrams (automata theory), Markov chains (probability theory), PERT networks (management decisions), games (artificial intelligence, mathematics), data structures (computer science), flow charts (chemistry, programming), crystal structures (physics), bonding structures (chemistry), transportation networks (operation research), family trees (genealogical theory), computer system configuration (computer architecture), semantic networks (artificial intelligence), augmented transition networks (artificial intelligence, linguistics), neural networks (neurophysiology, cybernetics), and phrase markers (linguistics) are some examples.

GRASPER 1.0 is a programming language extention 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.

The GRASPER-GRAPH data type supported by GRASPER 1.0 includes nodes, edges, spaces, and values. Nodes, edges, and spaces all have names and values. Edges are directed connections between pairs of nodes. Spaces are subsets of nodes, edges, and values, i.e., subgraphs. GRASPER 1.0 primitives are divided into three groups: Group I primitives apply to units of GRASPER-GRAPHs; Group II primitives apply to major portions of GRASPER-GRAPHs; Group III primitives pertain to memory management for GRASPER-GRAPH storage.

GRASPER 1.0 (GRASPe Extended and Revised) is both an extention and revision of GRASPE 1.5 [PRA71]. Like GRASPE 1.5, GRASPER 1.0 is a set of functions and pseudo-functions that could potentially be appended to any list processing system; however, particular emphasis is placed on

LISP 1.5 [McC65] as the host language. The syntax utilized in this manual is that of LISP 1.5.

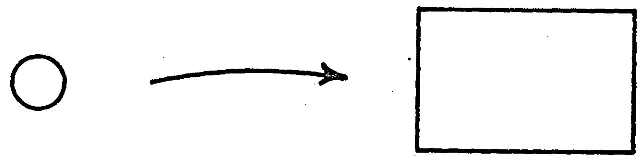
Throughout the rest of this manual "GRASPER 1.0" is abbreviated "GRASPER."

GRASPER-GRAPHS

GRASPER supports a specialized data type which will be referred to in this manual as a GRASPER-GRAPH, or simply a GRAPH. GRASPER provides facilities for the construction, destruction, and interrogation of GRASPER-GRAPHs. Understanding GRASPER-GRAPHs is a prerequisite to understanding GRASPER. This chapter is devoted to their definition.

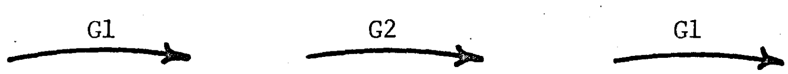
Informal Definition

1. A GRASPER-GRAPH consists of a collection of nodes, directed edges, and spaces.

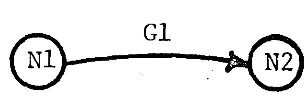


2. Each node has a unique name (an S-expression).

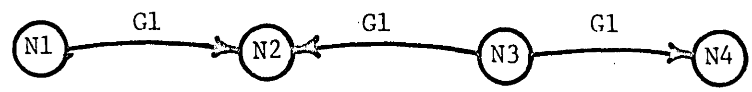
3. Edges also have names (S-expressions) but they are not necessarily unique.



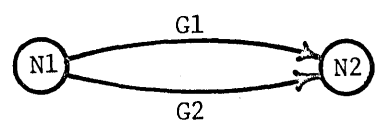
4. Each edge connects a pair of nodes.



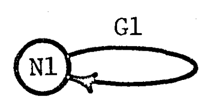
5. When two or more edges with the same name and direction leave a node, then each edge must point to a different node.



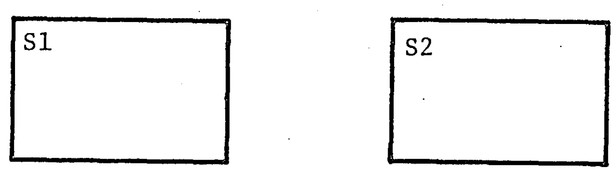
6. When two or more edges leave a node and both point to the same node, then each must have a different name.



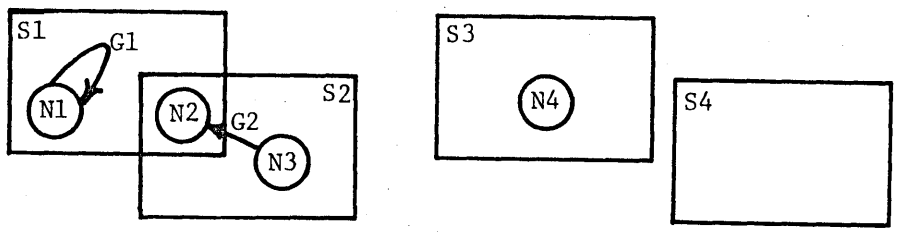
7. An edge may leave a node and point back to the same node.



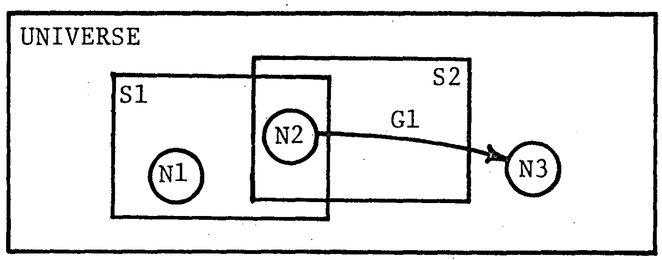
8. Spaces have unique names (S-expressions).



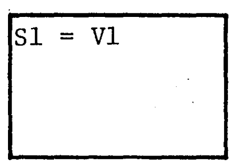
9. Each space contains a subset of all the nodes and edges. A node can be in any space. An edge can be in a space only if the pair of nodes it connects are also in that space.



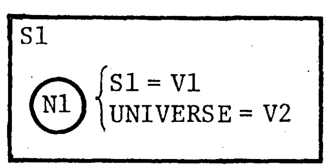
10. The universal space, labeled UNIVERSE, always contains all existing nodes and edges. The removal of a node or edge from UNIVERSE constitutes its removal from the system.



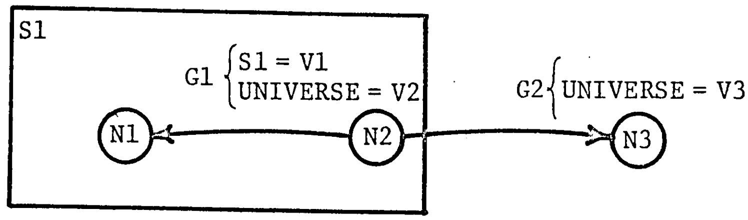
11. Each space has a value (an S-expression, initially NIL).



12. Each node has a value for each space it is in (an S-expression, initially NIL).



13. Each edge has a value for each space it is in (an S-expression, initially NIL).



Formal Definition

Let

$G = \{g \mid g \text{ is an S-expression}\};$

Each element of G represents a possible edge.

$V = \{v \mid v \text{ is an S-expression}\};$

Each element of V represents a possible value.

then

GRASPER-GRAPH = (N NGN S NSV NGNSV SV)[1]

where

$N = \{n \mid n \text{ is an S-expression}\},$
N is finite;

Each element of N represents an existing node.

$NGN \subseteq N \times G \times N;$

Each element of NGN represents an existing edge from the first node to the second node.

$S = \{s \mid s \text{ is an S-expression}\} \cup \{\text{UNIVERSE}\},$
S is finite;

Each element of S represents an existing space.

$NSV \subseteq NS \times V,$
$\{(n \text{ UNIVERSE}) \mid n \in N\} \subseteq NS \subseteq N \times S,$
for each $ns \in NS$ there is exactly one $v \in V$
s.t. $(ns\ v) \in NSV;$

Each element of NSV represents the existence of a node in a space and its value in that space.

[1] Although (NSV NGNSV SV) would be sufficient, the six-tuple is used to simplify the formal definitions of some GRASPER primitives.

$NGNSV \subseteq NGNS \times V$,

$\{(ngn\ UNIVERSE) | ngn \in NGN\} \subseteq NGNS \subseteq NGN \times S$,

for each $((n\ g\ m)\ s) \in NGNS$

$\exists v_1, v_2$ s.t. $((n\ s)\ v_1) \in NSV$, $((m\ s)\ v_2) \in NSV$,

for each $ngns \in NGNS$ there is exactly one $v \in V$

s.t. $(ngns\ v) \in NGNSV$;

Each element of NGNSV represents the existence of an edge in a space and its value in that space. Note that an edge can be in a space only if both of the nodes it connects are in that space.

$SV \subseteq S \times V$,

for each $s \in S$ there is exactly one $v \in V$

s.t. $(s\ v) \in SV$;

Each element of SV represents the value of a space.

Drawings of GRASPER-GRAPHs

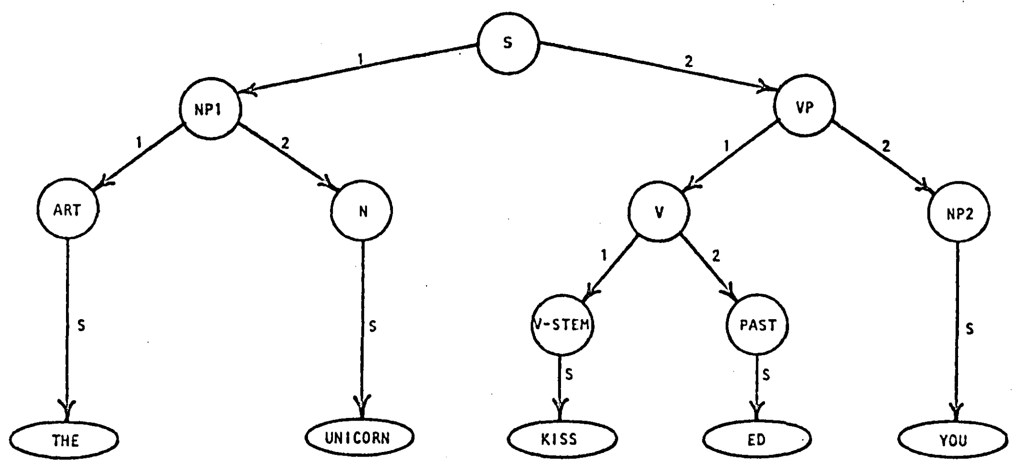
Throughout this manual, GRASPER-GRAPHs are drawn basically as in their informal definition. Nodes are drawn as circles or elipses, edges as arrows, and spaces as rectangles. Nodes and edges are in a space provided they are completely contained within the rectangle representing the space. The universal space, UNIVERSE, is often excluded from drawings since it is always implicitly there. When the values of GRASPER entities are of interest, they are drawn as in the informal definition. Since node and edge values in UNIVERSE can be viewed as global values, they are sometimes drawn N = val and $\xrightarrow{G\ =\ val}$.

This particular style of pictorially representing GRASPER-GRAPHs is not necessarily the best representation for a particular case. For example, spaces might be better represented by nodes and edges of different shapes or colors. Values of graph entities might be better displayed in a tabular format. The particular representation utilized should reflect the characteristics of the domain of application.

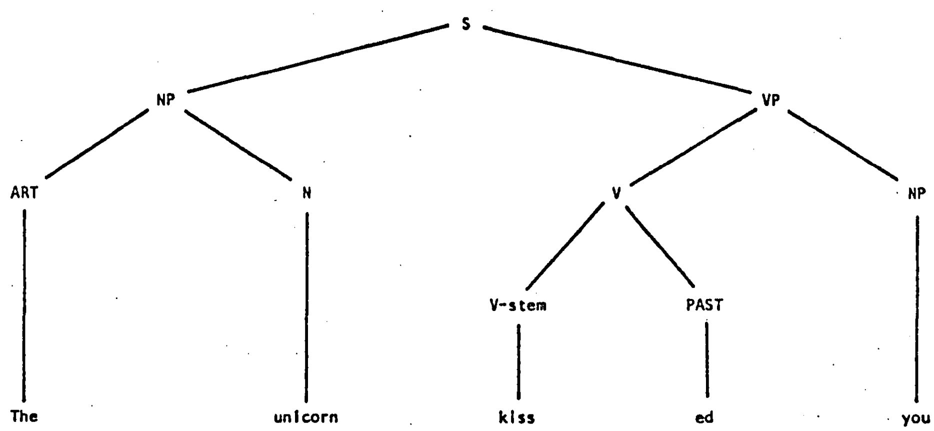
The reader is encouraged to study the following GRASPER-GRAPHs since they are utilized throughout the remainder of this manual.

Illustrations

Phrase Marker

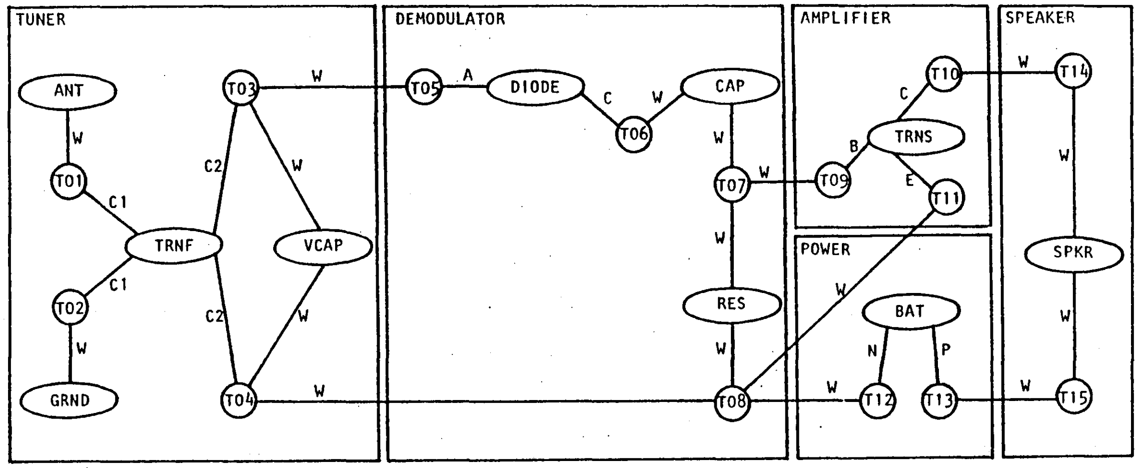


This GRAPH represents the following phrase marker.

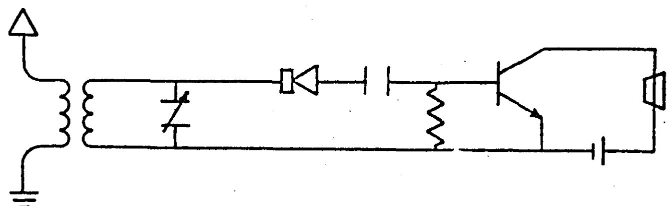


The direction and labeling of edges in the GRAPH are used to incorporate the information implicit in the position of the terminals and nonterminals in the phrase marker. All edges point (down) towards the surface. Numbers on edges order the nonterminals to which they point. S-edges point to surface terminals.

Radio



This GRAPH represents the following schematic for a radio.

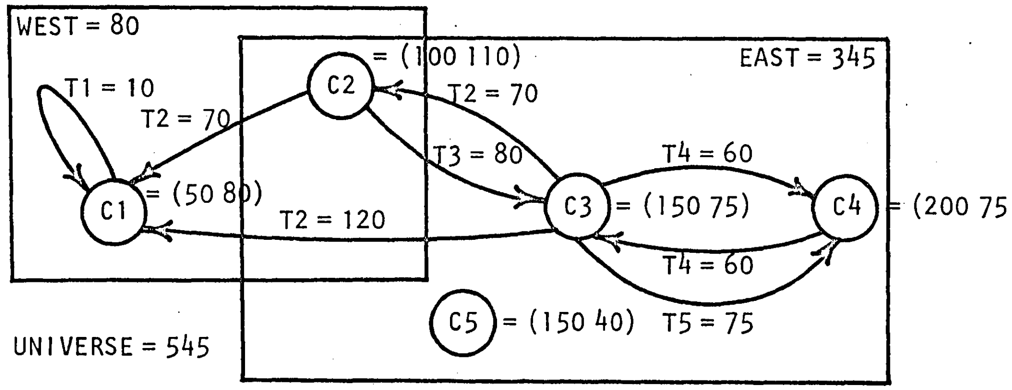


The labels on the GRAPH translate as follows.

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

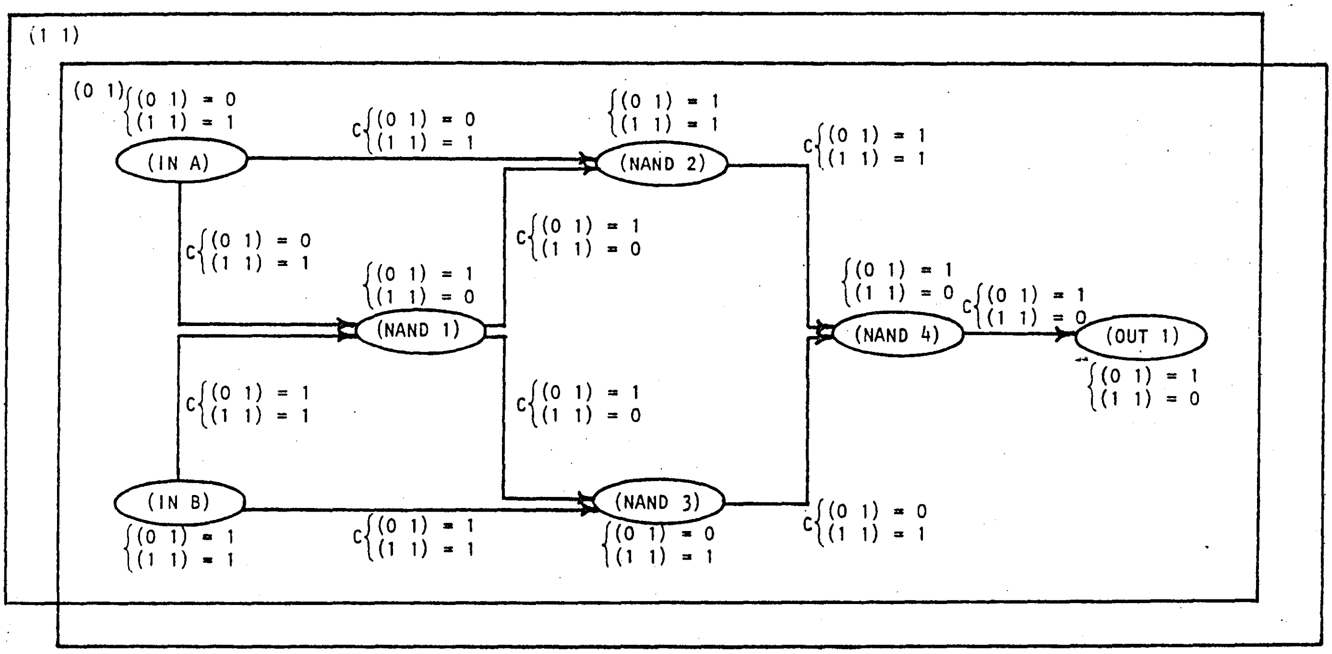
The undirected edges indicate two edges labeled the same with opposite direction. The spaces indicate the major components of the radio.

Railroad

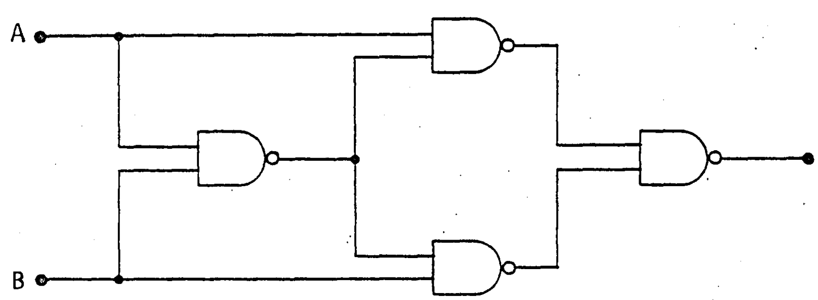


The above GRAPH represents a railroad's system of tracks servicing five cities. Each city is represented by a node. Tracks are represented by edges between cities. Commuter tracks local to a city are represented as edges which originate and end at that city. The direction of each edge indicates the direction 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.

Xor

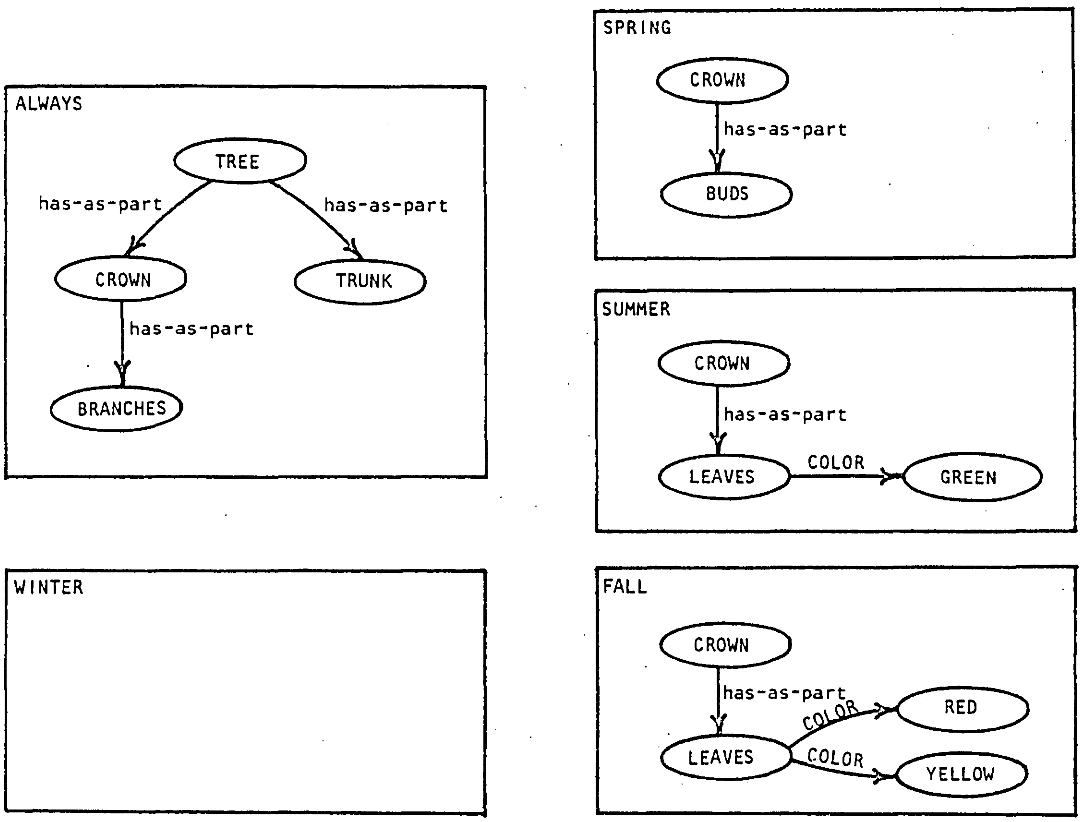


This GRAPH represents two states of the following logic diagram for exclusive or.



The values in space (0 1) indicate the state of the components when input A is 0 and input B is 1. The values in space (1 1) correspond to the state when both inputs are 1.

Tree



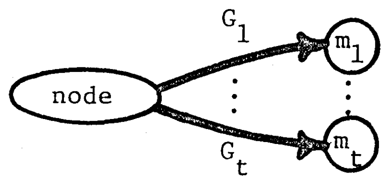
The above GRAPH is a semantic network describing trees over the seasons. The space ALWAYS contains information about trees that is true during all seasons. Each of the other spaces contains information about trees that is true in the corresponding season.

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).

Additional GRASPER-GRAPH Terminology

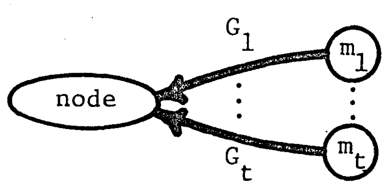
Outpointing Edges

All edges which point away from a node are said to be outpointing with respect to that node.



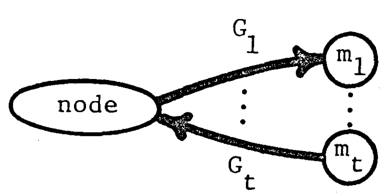
Inpointing Edges

All edges which point to a node are said to be inpointing with respect to that node.



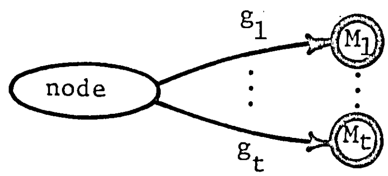
Adjacent Edges

All edges which are connected to a node are said to be adjacent to that node.



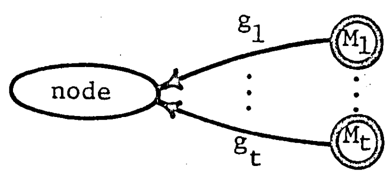
Outpointing Nodes

All nodes pointed to by the outpointing edges of a node are said to be outpointing with respect to that node.



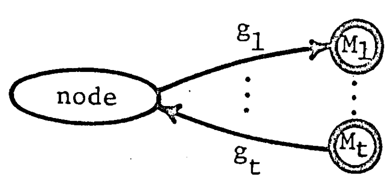
Inpointing Nodes

All nodes from which the inpointing edges of a node originate are said to be inpointing with respect to that node.



Adjacent Nodes

All nodes connected to the other ends of the adjacent edges of a node are said to be adjacent with respect to that node.

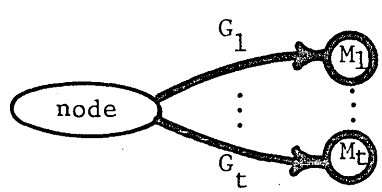


Pairs

A pair consists of an edge and a node. A pair coupled with a node and qualifying direction (outpointing or inpointing) is used in GRASPER to uniquely identify an edge. This is required since edge names are not necessarily unique.

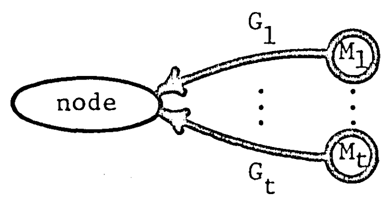
Outpointing Pairs

All pairs where each consists of an outpointing edge of a node and the node to which it points are said to be outpointing with respect to that node.



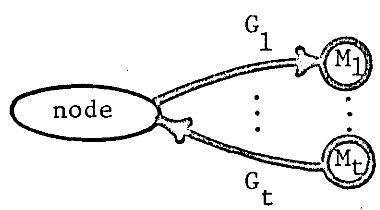
Inpointing Pairs

All pairs where each consists of an inpointing edge of a node and the node from which it originates are said to be inpointing with respect to that node.



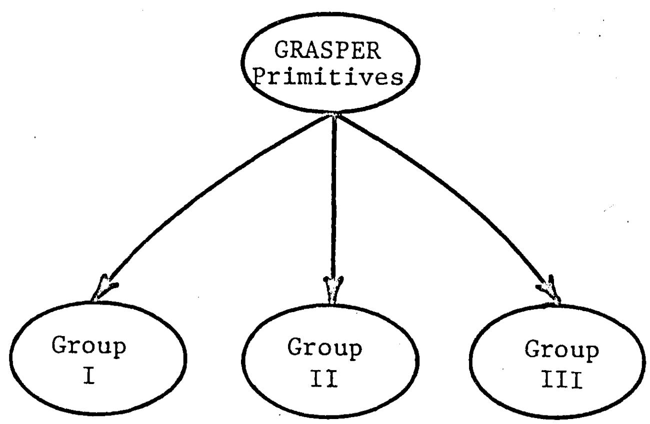
Adjacent Pairs

All pairs where each consists of an adjacent edge of a node and the node at the other end of the edge are said to be adjacent with respect to that node.



GRASPER PRIMITIVES

GRASPER primitives are divided into three groups. Group I primitives manipulate units of GRASPER-GRAPHs. They form the basis of the system. Group II primitives are concerned with the destruction, creation, and description of major portions of GRAPHs. Group III primitives pertain to memory management.



Group I Primitives

The following rules describe how the names of Group I operators are composed from more primitive GRASPER concepts. The next section contains a complete description of each composable operator. The reader is encouraged to refer to these complete descriptions while reading the rules of operator composition.

Rules of Group I Operator Composition

1. The name of each Group I primitive operator is formed by concatenating three or four single letter abbreviations for GRASPER concepts.

   e.g., $C \oplus O \oplus P = COP$
   $S \oplus I \oplus N \oplus G = SING$

2. The role played by each letter is determined by its position in the operator's name as follows.

   <operator type><object qualifier><operator object>
   <operator type><object qualifier><operator object><qualifying object>

3. There are two major categories of operators, functions and pseudo-functions. Functions are executed for the value they return. Pseudo-functions are executed for their effect. Group I pseudo-functions all return their first argument. Each of these categories include three operator types as defined below. The single letter abbreviation for each is underlined.

pseudo-functions	Create	= creates the specified GRAPH entity(s) if it does not already exist
	Destroy	= destroys the specified GRAPH entity(s) if it exists
	Bind	= binds the specified GRAPH entity(s) to the specified value
functions	Set of	= returns the set of specified GRAPH entities
	Value of	= returns the value to which the specified GRAPH entity(s) is bound
	eXistence of	= returns T if the specified GRAPH entity(s) exists, and NIL if it does not

4. The operator object specifies the type of GRAPH entity(s) in the current GRAPH the operator manipulates. These include the following (their abbreviations are underlined).

Space(s)	edGe(s)
Node(s)	Pair(s)

e.g., D␣N destroys nodes
S␣N returns a set of nodes

5. Object qualifiers include the following types (their abbreviations are underlined).

Outpointing	Adjacent
Inpointing	Unqualified

6. Outpointing, inpointing, and adjacent qualifiers specify the means of accessing the object(s) from a given node (see pages 16-19). Unqualified is used when access is immediate through the object's name (which must be unique). Therefore,

- edges and pairs must be qualified
- nodes may be qualified or unqualified
- spaces are always unqualified

$\sqcup\begin{matrix}O\\I\\A\end{matrix}G \qquad \sqcup\begin{matrix}O\\I\\A\end{matrix}P \qquad \sqcup\begin{matrix}O\\I\\A\end{matrix}N \qquad \sqcup UN \qquad \sqcup US$

e.g., DOG - destroys the outpointing edges of a node
CUS - creates a space
SAN - returns the adjacent nodes of a node
VUN - returns the value of a node

7. Qualified edges and qualified nodes can be followed by <u>qualifying objects</u>. These indicate additional arguments which restrict the objects of the operator. The following combinations are possible.

```
 O G        O
⌴IGN      ⌴ING
 A          A
```

e.g., DOGN - destroys the outpointing edges of a node which lead to a given node

SANG - returns the adjacent nodes of a node which are connected by a given edge

Note the difference between these examples and the analogous ones for rule 6.

8. C-, B-, V-, and X-type operators create, bind, return the value of and test the existence of only uniquely specified entities. Therefore, they only take (qualified) pairs, unqualified nodes, and (unqualified) spaces as objects.

9. "SOGG", "SIGG", and "SAGG" are <u>not included</u> since they would serve no useful purpose. All would either return the empty set or the set consisting of the given edge as its only element. SONG, SING, and SANG return that information and more.

10. Arguments of operators:

   a) All operators with a qualified object have a node as their first argument. This is the node to which the qualifier refers.

   b) All operators, other than S-types, with an unqualified object or a pair as their object have the object as their next argument. If it is a pair the parentheses around the pair are dropped, thus the edge and node are the next two arguments.

      S-type operators do not have their objects as an argument since it is a set of those objects that is to be returned. Operators with a qualified object other than a pair do not have the object as an argument since they specify their object(s) through relative position rather than by name.

   c) All operators with a qualifying object have it as their next argument.

   d) All B-operators have the value to which the object is to be bound as their next argument.

   e) All operators, except those whose objects are spaces, have an optional space as their next argument. When this argument is not included, it is assumed to be UNIVERSE. This argument indicates the space in which the object is to be affected or sought.

   f) SUS is a special case. It has an optional node as its only argument.

On the following three pages are two views of all Group I operators.

Group I GRASPER Operators: Tabular Summary

(B$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$ *value*)(B$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$ *value space*)

(BUN *node value*)(BUN *node value space*)

(BUS *space value*)

(C$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$)(C$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$ *space*)

(CUN *node*)(CUN *node space*)

(CUS *space*)

(D$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$)(D$\overset{O}{\underset{A}{I}}$P *node*$_1$ *edge* *node*$_2$ *space*)

(D$\overset{O}{\underset{A}{I}}$G *node*)(D$\overset{O}{\underset{A}{I}}$G *node space*)

(D$\overset{O}{\underset{A}{I}}$GG *node edge*)(D$\overset{O}{\underset{A}{I}}$GG *node edge space*)

(D$\overset{O}{\underset{A}{I}}$GN *node*$_1$ *node*$_2$)(D$\overset{O}{\underset{A}{I}}$GN *node*$_1$ *node*$_2$ *space*)

(D$\overset{O}{\underset{A}{I}}$N *node*)(D$\overset{O}{\underset{A}{I}}$N *node space*)

(D$\overset{O}{\underset{A}{I}}$NG *node edge*)(D$\overset{O}{\underset{A}{I}}$NG *node edge space*)

(DUN *node*)(DUN *node space*)

(DUS *space*)

(S$\underset{A}{\overset{O}{\text{I}}}$P *node*)(S$\underset{A}{\overset{O}{\text{I}}}$P *node space*)

(S$\underset{A}{\overset{O}{\text{I}}}$G *node*)(S$\underset{A}{\overset{O}{\text{I}}}$G *node space*)

(S$\underset{A}{\overset{O}{\text{I}}}$GN *node*$_1$ *node*$_2$)(S$\underset{A}{\overset{O}{\text{I}}}$GN *node*$_1$ *node*$_2$ *space*)

(S$\underset{A}{\overset{O}{\text{I}}}$N *node*)(S$\underset{A}{\overset{O}{\text{I}}}$N *node space*)

(S$\underset{A}{\overset{O}{\text{I}}}$NG *node edge*)(S$\underset{A}{\overset{O}{\text{I}}}$NG *node edge space*)

(SUN)(SUN *space*)

(SUS)(SUS *node*)

(V$\underset{A}{\overset{O}{\text{I}}}$P *node*$_1$ *edge node*$_2$)(V$\underset{A}{\overset{O}{\text{I}}}$P *node*$_1$ *edge node*$_2$ *space*)

(VUN *node*)(VUN *node space*)

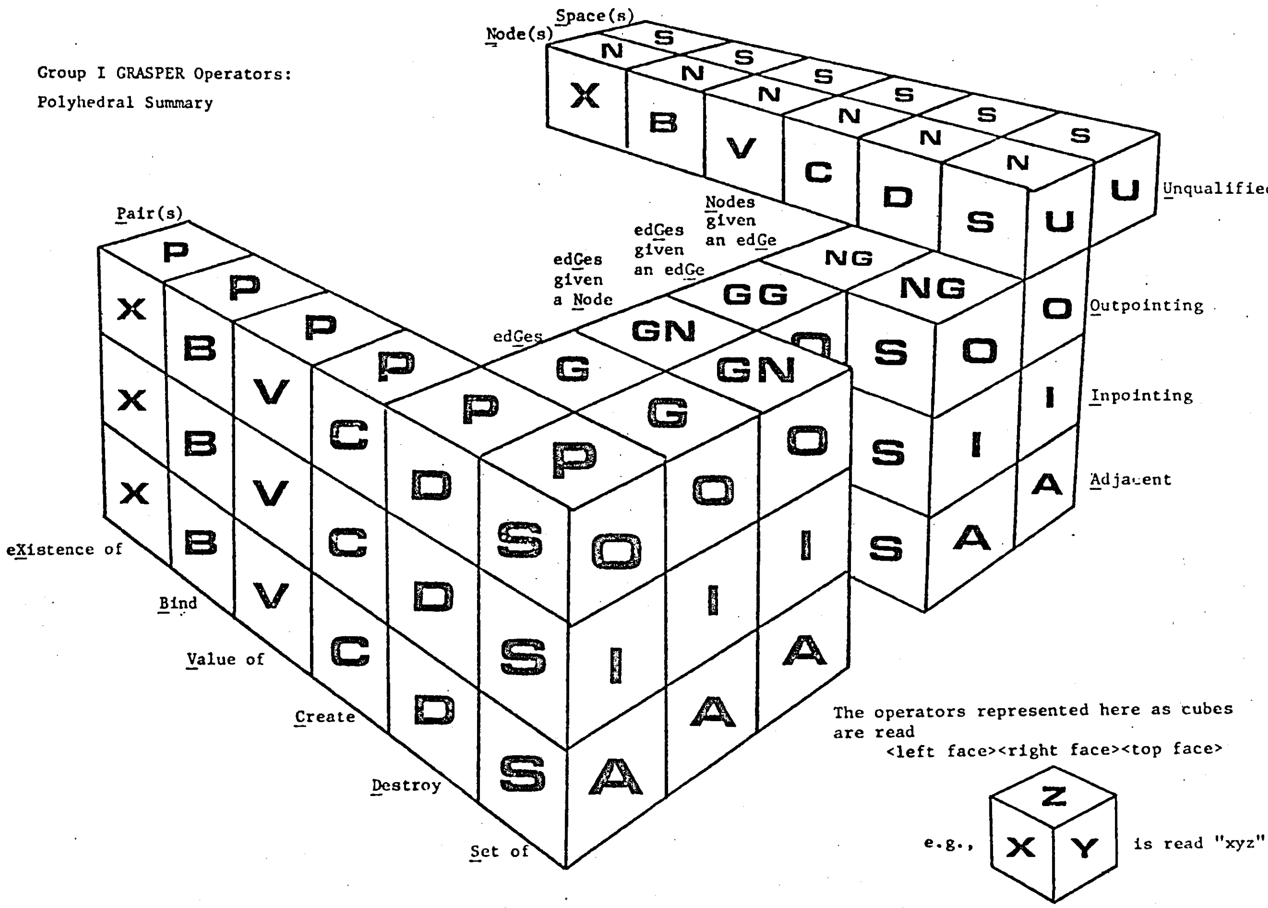
(VUS *space*)

(X$\underset{A}{\overset{O}{\text{I}}}$P *node*$_1$ *edge node*$_2$)(X$\underset{A}{\overset{O}{\text{I}}}$P *node*$_1$ *edge node*$_2$ *space*)

(XUN *node*)(XUN *node space*)

(XUS *space*)

Group I GRASPER Operators: Polyhedral Summary



Group I Operator Descriptions

This section contains a complete description of each (legally composable) Group I operator in alphabetical order. Each description consists of

1) the calling form (in LISP syntax) of the operator with its arguments,
2) the derivation of its acronym,
3) its pronunciation key[1],
4) its informal definition including
   (a) a prose description of the operator's purpose,
   (b) a graphical description of its purpose (bold lines are used to indicate newly constructed entities or entities returned, and broken lines are used to indicate deleted entities),
   and (c) a prose description of each GRASPER error condition,
5) its formal definition including all GRASPER error conditions,

and 6) a group of illustrations (in LISP syntax) including the generation of each GRASPER error condition.

Each group of illustrations begins with a drawing of the GRAPH which exists before each illustrative call. The series of calls does not represent a sequence during one user session. If a call alters the GRAPH, a drawing of the resulting GRAPH is given.

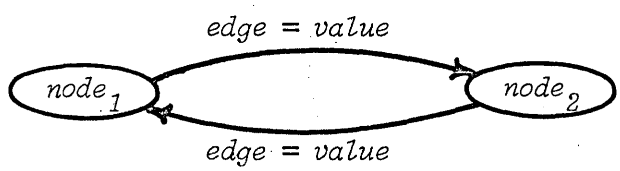
The descriptions follow in alphabetical order.

[1] See APPENDIX A on page 343 for a guide to the pronunciation symbols.

(BAP $node_1$ $edge$ $node_2$ $value$)[1] Bind Adjacent Pairs \'bap\

Informal Definition

The pseudo-function BAP is an EXPR which has the effect of binding the adjacent pairs ($edge$ $node_2$) of $node_1$ to $value$ in the universal space. Given $node_1$, $edge$, $node_2$, and $value$, BAP binds $edge$ pointing from $node_1$ to $node_2$ and/or $edge$ pointing from $node_2$ to $node_1$ to $value$ in UNIVERSE. If the edges are already bound to $value$, BAP has no effect. BAP returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist
- ($edge$ $node_2$) is neither an outpointing nor inpointing pair of $node_1$

Formal Definition

BAP[n,g,m,v] = n

with effects:

if XOP[n,g,m,v] = T then BOP[n,g,m,v]

if XIP[n,g,m,v] = T then BIP[n,g,m,v]

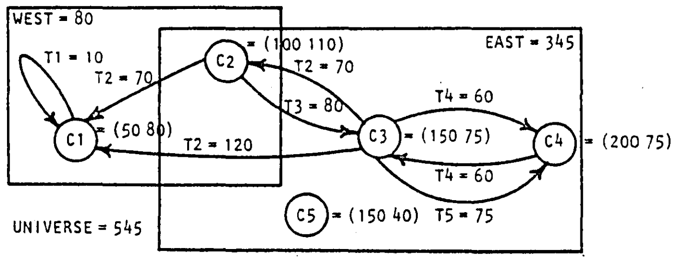
error conditions:

- $n \notin N$
- $m \notin N$
- $(n\ g\ m) \notin NGN$ and $(m\ g\ n) \notin NGN$

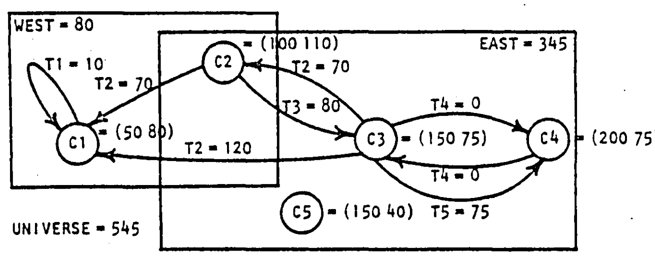
---

[1] See alternative form on page 32.

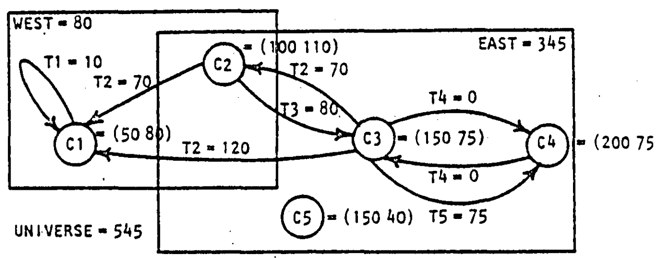
Illustrations



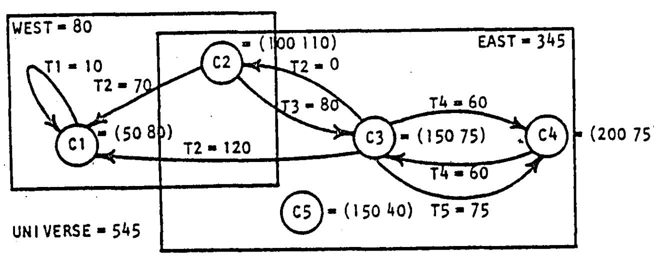
?(BAP 'C3 'T4 'C4 0)
C3



?(BAP 'C4 'T4 'C3 0)
C4



?(BAP 'C3 'T2 'C2 0)
C3



?(BAP 'C3 'T4 'C4 60)
C3

?(BAP 'C3 'TX 'C4 0)

*** BAP ERROR: THERE IS NO EDGE TX BETWEEN NODE C3 AND NODE C4

?(BAP 'CX 'T4 'C4 0)

*** BAP ERROR: CX IS NOT A NODE

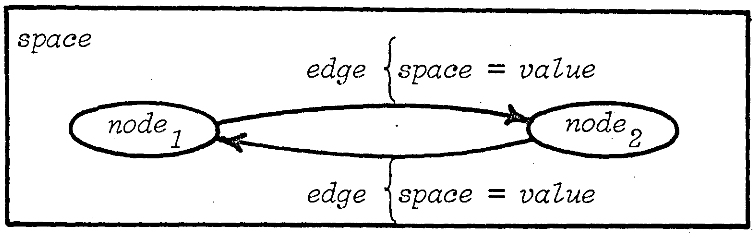
?(BAP 'C3 'T4 'CX 0)

*** BAP ERROR: CX IS NOT A NODE

(BAP *node*$_1$ *edge* *node*$_2$ *value* *space*)[1] <u>B</u>ind <u>Ad</u>jacent <u>P</u>áirs \'bap\

Informal Definition

The pseudo-function BAP is an EXPR which has the effect of binding the adjacent pairs (*edge* *node*$_2$) of *node*$_1$ to *value* in *space*. Given *node*$_1$, *edge*, *node*$_2$, *value*, and *space*, BAP binds *edge* pointing from *node*$_1$ to *node*$_2$ and/or *edge* pointing from *node*$_2$ to *node*$_1$ to *value* in *space*. If the edges are already bound to *value* in *space*, BAP has no effect. BAP returns *node*$_1$.



error conditions:

- *node*$_1$ does not exist in *space*
- *node*$_2$ does not exist in *space*
- (*edge* *node*$_2$) is neither an outpointing nor inpointing pair of *node*$_1$ in *space*
- *space* does not exist

Formal Definition

BAP[n,g,m,v,s] = n

with effects:

if XOP[n,g,m,s] = T then BOP[n,g,m,v,s]

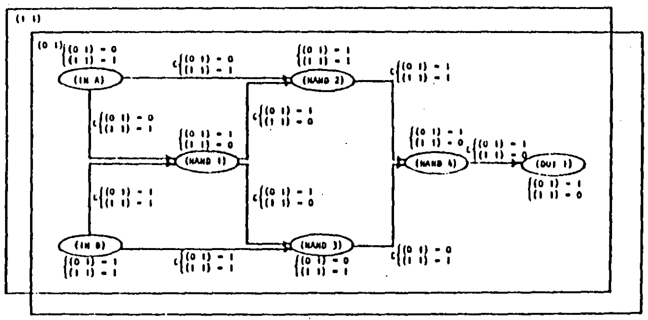
if XIP[n,g,m,s] = T then BIP[n,g,m,v,s]

error conditions:

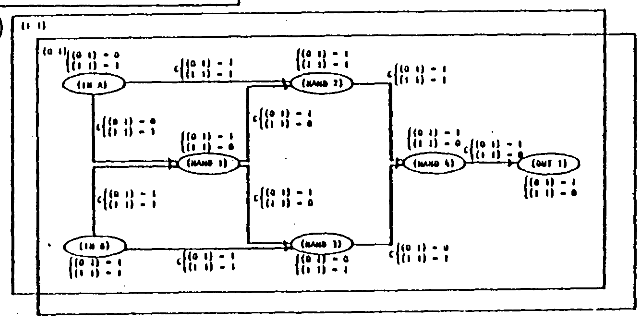
- ((n s) v') $\notin$ NSV for all v' $\in$ V
- ((m s) v') $\notin$ NSV for all v' $\in$ V
- (((n g m) s) v') $\notin$ NGNSV for all v' $\in$ V and (((m g n) s) v') $\notin$ NGNSV for all v' $\in$ V
- s $\notin$ S

[1]See alternative form on page 30.

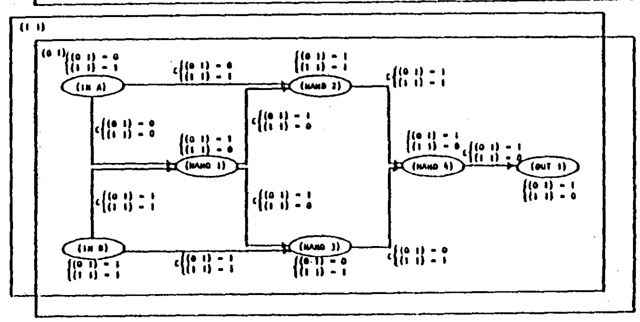
Illustrations



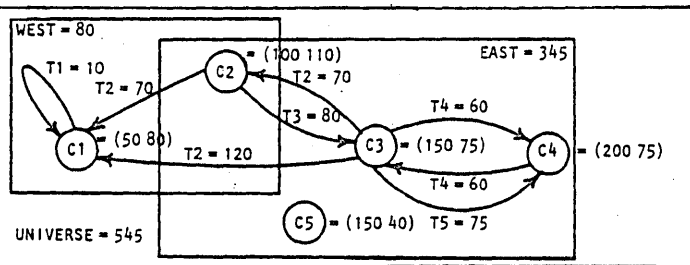
```
?(BAP '(IN A) 'C '(NAND 2) 1 '(0 1))
(IN A)
```



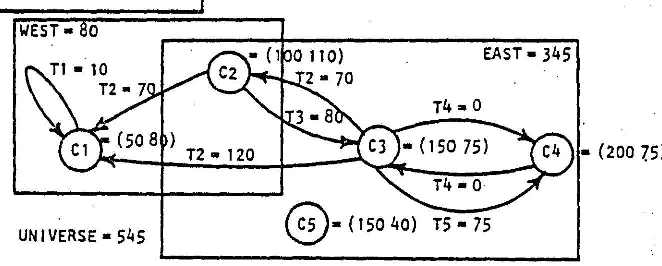
```
?(BAP '(NAND 1) 'C '(IN A) 0 '(1 1))
(NAND 1)
```



```
?(BAP '(IN A) 'C '(NAND 2) 0 '(0 1))
(IN A)
```




```
?(BAP 'C3 'T4 'C4 0 'UNIVERSE)
C3
```

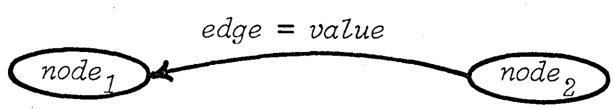



```
?(BAP 'C3 'TX 'C4 0 'EAST)
*** BAP ERROR: THERE IS NO EDGE TX BETWEEN NODE C3 AND NODE C4 IN SPACE EAST

?(BAP 'CX 'T4 'C4 0 'EAST)
*** BAP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BAP 'C3 'T4 'CX 0 'EAST)
*** BAP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BAP 'C3 'T4 'C4 0 'SX)
*** BAP ERROR: SX IS NOT A SPACE
```

(BIP $node_1$ *edge* $node_2$ *value*)[1] <u>B</u>ind <u>I</u>npointing <u>P</u>air \'bip\

<u>Informal Definition</u>

The pseudo-function BIP is an EXPR which has the effect of binding the inpointing pair (*edge* $node_2$) of $node_1$ to *value* in the universal space. Given $node_1$, *edge*, $node_2$, and *value*, BIP binds *edge* pointing from $node_2$ to $node_1$ to *value* in UNIVERSE. If the edge is already bound to *value*, BIP has no effect. BIP returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist
- (*edge* $node_2$) is not an inpointing pair of $node_1$

<u>Formal Definition</u>

BIP[n,g,m,v] = n

with effects:

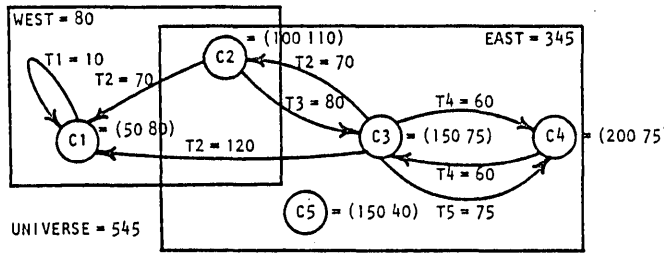
$$\text{NGNSV} := (\text{NGNSV} - \{(((m\ g\ n)\ \text{UNIVERSE})\ v') \mid v' \in V\}) \cup \{(((m\ g\ n)\ \text{UNIVERSE})\ v)\}$$

error conditions:

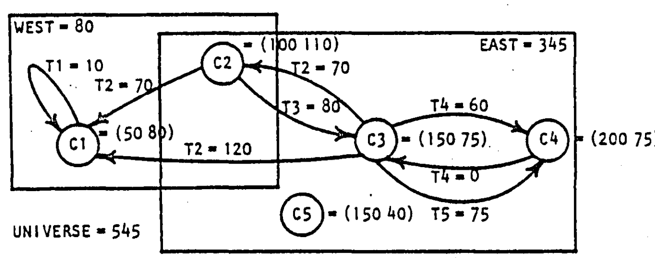
- $n \notin N$
- $m \notin N$
- $(m\ g\ n) \notin NGN$

[1] See alternative form on page 36.

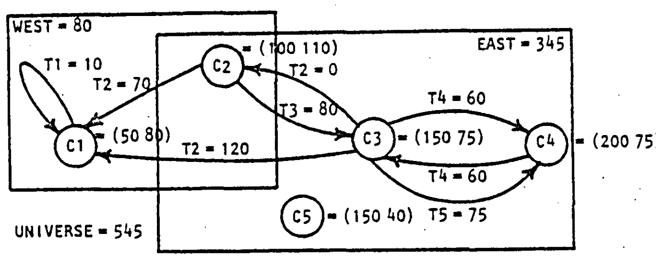
Illustrations




```
?(BIP 'C3 'T4 'C4 0)
C3
```




```
?(BIP 'C2 'T2 'C3 0)
C2
```

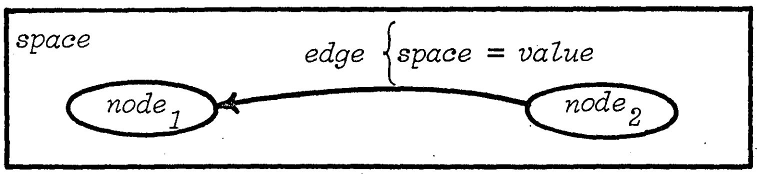



```
?(BIP 'C2 'T2 'C3 70)
C2

?(BIP 'C2 'TX 'C3 0)
*** BIP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C3 TO NODE C2

?(BIP 'CX 'T2 'C3 0)
*** BIP ERROR: CX IS NOT A NODE

?(BIP 'C2 'T2 'CX 0)
*** BIP ERROR: CX IS NOT A NODE
```

(BIP $node_1$ *edge* $node_2$ *value space*) Bind Inpointing Pair \'bip\

Informal Definition

The pseudo-function BIP is an EXPR which has the effect of binding the inpointing pair (*edge* $node_2$) of $node_1$ to *value* in *space*. Given $node_1$, *edge*, $node_2$, *value*, and *space*, BIP binds *edge* pointing from $node_2$ to $node_1$ to *value* in *space*. If the edge is already bound to *value* in *space*, BIP has no effect. BIP returns $node_1$.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- (*edge* $node_2$) is not an inpointing pair of $node_1$ in *space*
- *space* does not exist

Formal Definition

BIP[n,g,m,v,s] = n

with effects:

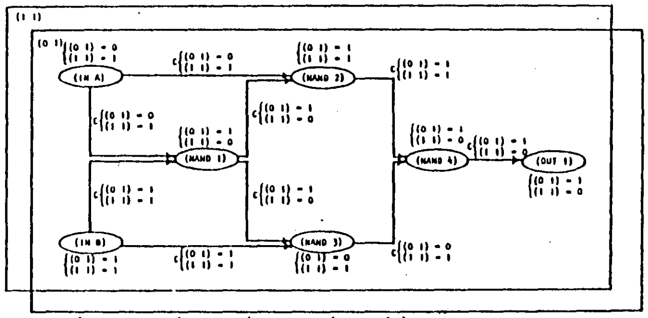
$$NGNSV := (NGNSV - \{(((m\ g\ n)\ s)\ v') \mid v' \in V\}) \cup \{(((m\ g\ n)\ s)\ v)\}$$

error conditions:

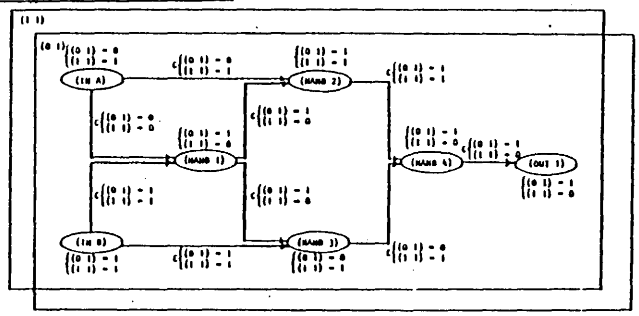
- $((n\ s)\ v') \notin NSV$ for all $v' \in V$
- $((m\ s)\ v') \notin NSV$ for all $v' \in V$
- $(((m\ g\ n)\ s)\ v') \notin NGNSV$ for all $v' \in V$
- $s \notin S$

[1]See alternative form on page 34.

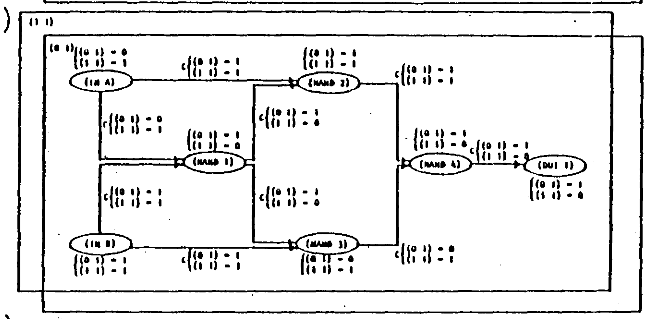
Illustrations



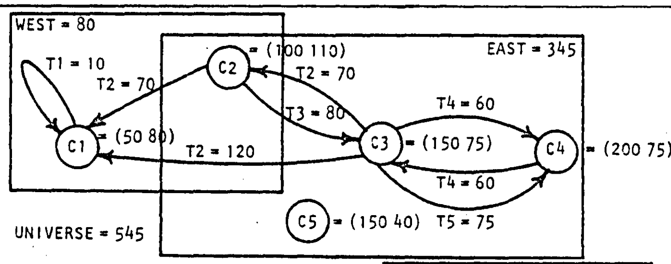
```
?(BIP '(NAND 1) 'C '(IN A) 0 '(1 1))
(NAND 1)
```



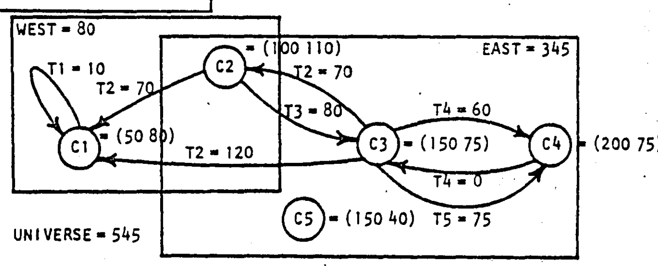
```
?(BIP '(NAND 2) 'C '(IN A) 1 '(0 1))
(NAND 2)
```



```
?(BIP '(NAND 2) 'C '(IN A) 0 '(0 1))
(NAND 2)
```




```
?(BIP 'C3 'T4 'C4 0 'UNIVERSE)
C3
```

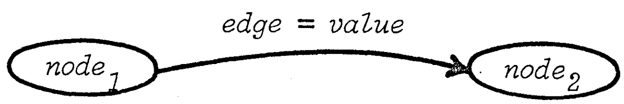



```
?(BIP 'C3 'TX 'C4 0 'EAST)
*** BIP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C4 TO NODE C3 IN SPACE EAST

?(BIP 'CX 'T4 'C4 0 'EAST)
*** BIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BIP 'C3 'T4 'CX 0 'EAST)
*** BIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BIP 'C3 'T4 'C4 0 'SX)
*** BIP ERROR: SX IS NOT A SPACE
```

(BOP *node*$_1$ *edge* *node*$_2$ *value*)[1] <u>B</u>ind <u>O</u>utpointing <u>P</u>air \'bäp\

<u>Informal Definition</u>

The pseudo-function BOP is an EXPR which has the effect of binding the outpointing pair (*edge* *node*$_2$) of *node*$_1$ to *value* in the universal space. Given *node*$_1$, *edge*, *node*$_2$, and *value*, BOP binds *edge* pointing from *node*$_1$ to *node*$_2$ to *value* in UNIVERSE. If the edge is already bound to *value*, BOP has no effect. BOP returns *node*$_1$.



error conditions:

- *node*$_1$ does not exist
- *node*$_2$ does not exist
- (*edge* *node*$_2$) is not an outpointing pair of *node*$_1$

<u>Formal Definition</u>

BOP[n,g,m,v] = n

with effects:

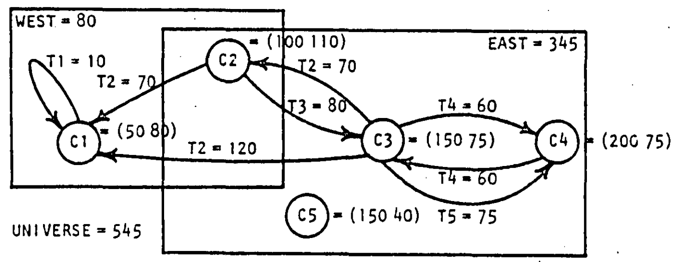
$$\text{NGNSV} := (\text{NGNSV} - \{(((n\ g\ m)\ \text{UNIVERSE})\ v') \mid v' \in V\}) \cup \{(((n\ g\ m)\ \text{UNIVERSE})\ v)\}$$

error conditions:

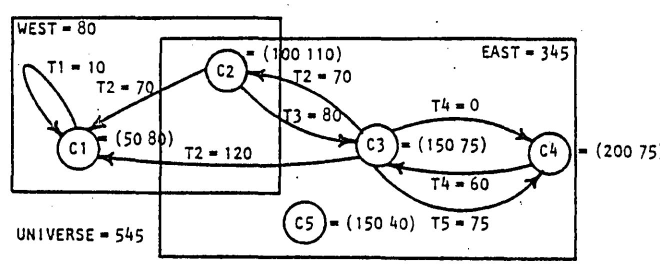
- $n \notin N$
- $m \notin N$
- $(n\ g\ m) \notin \text{NGN}$

[1]See alternative form on page 40.

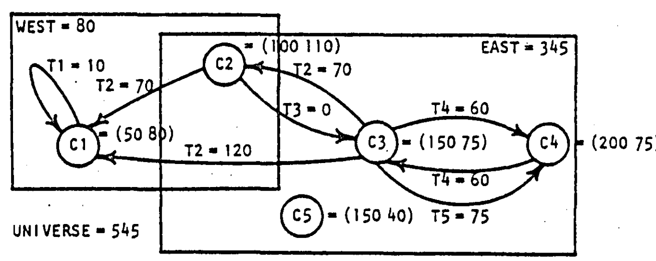
Illustrations




```
?(BOP 'C3 'T4 'C4 0)
C3
```




```
?(BOP 'C2 'T3 'C3 0)
C2
```

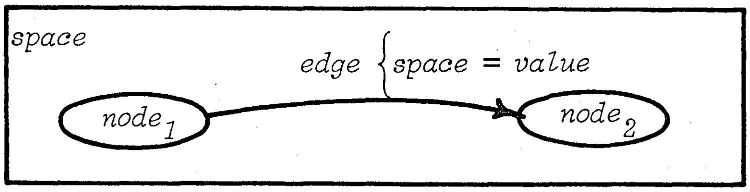



```
?(BOP 'C2 'T3 'C3 80)
C2

?(BOP 'C2 'TX 'C3 0)
*** BOP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C2 TO NODE C3

?(BOP 'C2 'T3 'CX 0)
*** BOP ERROR: CX IS NOT A NODE

?(BOP 'CX 'T3 'C3 0)
*** BOP ERROR: CX IS NOT A NODE
```

(BOP *$node_1$ edge $node_2$ value space*)[1] Bind Outpointing Pair \'bäp\

Informal Definition

The pseudo-function BOP is an EXPR which has the effect of binding the outpointing pair (*edge* $node_2$) of $node_1$ to *value* in *space*. Given $node_1$, *edge*, $node_2$, *value*, and *space*, BOP binds *edge* pointing from $node_1$ to $node_2$ to *value* in *space*. If the edge is already bound to *value* in *space*, BOP has no effect. BOP returns $node_1$.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- (*edge* $node_2$) is not an outpointing pair of $node_1$ in *space*
- *space* does not exist

Formal Definition

BOP[n,g,m,v,s] = n

with effects:

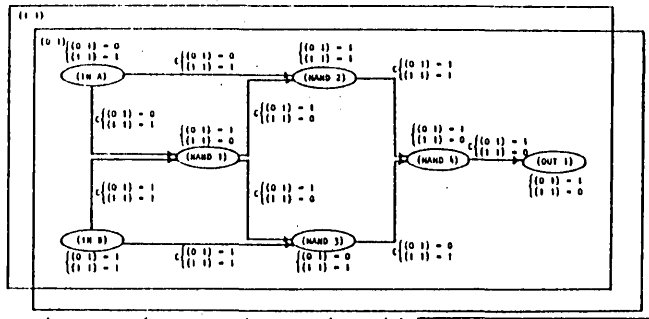
$$NGNSV := (NGNSV - \{(((n\ g\ m)\ s)\ v') \mid v' \in V\}) \cup \{(((n\ g\ m)\ s)\ v)\}$$

error conditions:

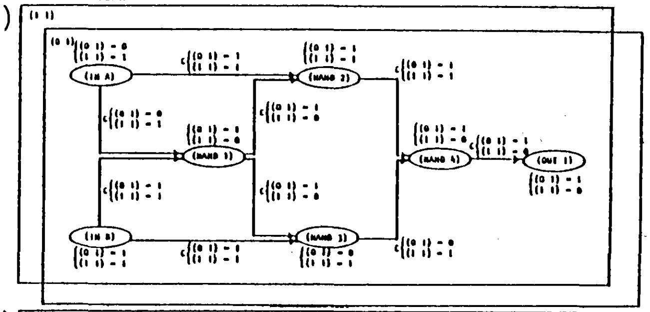
- $((n\ s)\ v') \notin NSV$ for all $v' \in V$
- $((m\ s)\ v') \notin NSV$ for all $v' \in V$
- $(((n\ g\ m)\ s)\ v') \notin NGNSV$ for all $v' \in V$
- $s \notin S$

[1] See alternative form on page 38.

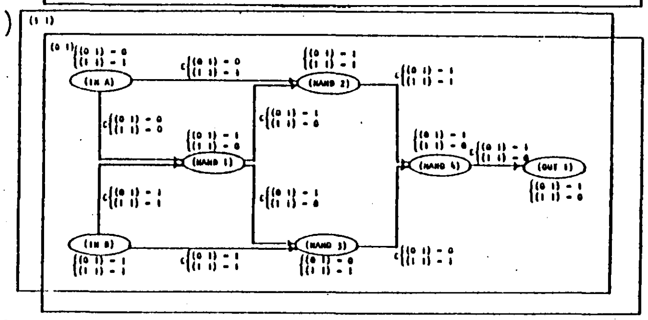
Illustrations



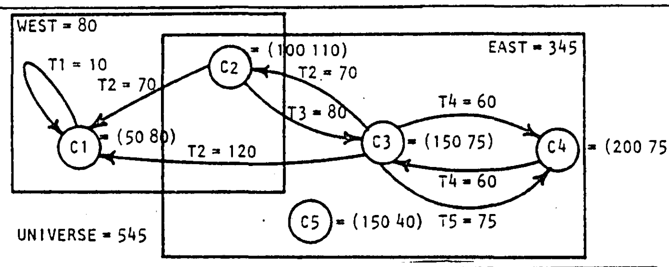
?(BOP '(IN A) 'C '(NAND 2) 1 '(0 1))
(IN A)



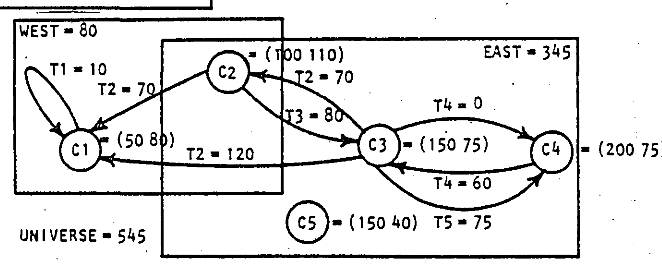
?(BOP '(IN A) 'C '(NAND 1) 0 '(1 1))
(IN A)



?(BOP '(IN A) 'C '(NAND 2) 0 '(0 1))
(IN A)



?(BOP 'C3 'T4 'C4 0 'UNIVERSE)
C3



?(BOP 'C3 'TX 'C4 0 'EAST)
*** BOP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C3 TO NODE C4 IN SPACE EAST

?(BOP 'CX 'T4 'C4 0 'EAST)
*** BOP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BOP 'C3 'T4 'CX 0 'EAST)
*** BOP ERROR: CX IS NOT A NODE IN SPACE EAST

?(BOP 'C3 'T4 'C4 0 'SX)
*** BOP ERROR: SX IS NOT A SPACE

(BUN *node value*)[1] Bind Unqualified Node \'bən\

Informal Definition

The pseudo-function BUN is an EXPR which has the effect of binding *node* to *value* in the universal space. Given *node* and *value*, BUN binds *node* to *value* in UNIVERSE. If *node* is already bound to *value* in *space*, BUN has no effect. BUN returns *node*.

node = *value*

error condition:

- *node* does not exist

Formal Definition

BUN[n,v] = n

with effects:

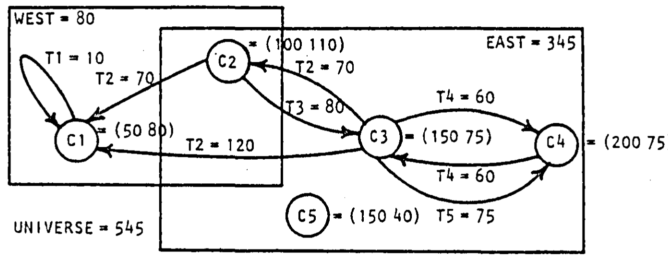
$$NSV := (NSV - \{((n\ UNIVERSE)\ v') \mid v' \in V\}) \cup \{((n\ UNIVERSE)\ v)\}$$

error condition:

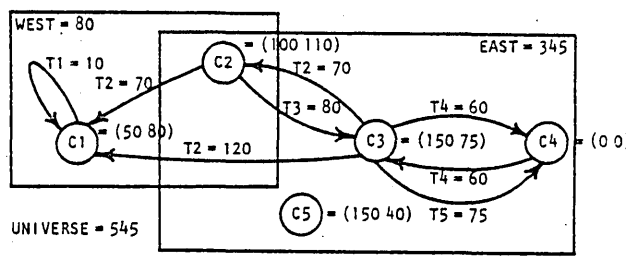
- $n \notin N$

[1]See alternative form on page 44.

Illustrations




```
?(BUN 'C4 '(0 0))
C4
```

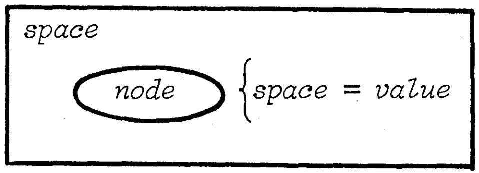



```
?(BUN 'C4 '(200 75))
C4

?(BUN 'CX '(0 0))
*** BUN ERROR: CX IS NOT A NODE
```

(BUN *node value space*)[1] Bind Unqualified Node \'bən\

Informal Definition

The pseudo-function BUN is an EXPR which has the effect of binding *node* to *value* in *space*. Given *node*, *value*, and *space*, BUN binds *node* to *value* in *space*. If *node* is already bound to *value* in *space*, BUN has no effect. BUN returns *node*.



error conditions:

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

Formal Definition

BUN[n,v,s] = n

with effects:

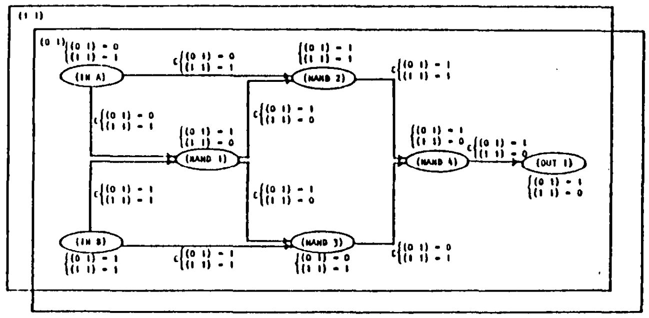
$$NSV := (NSV - \{((n\ s)\ v') \mid v' \in V\}) \cup \{((n\ s)\ v)\}$$

error conditions:

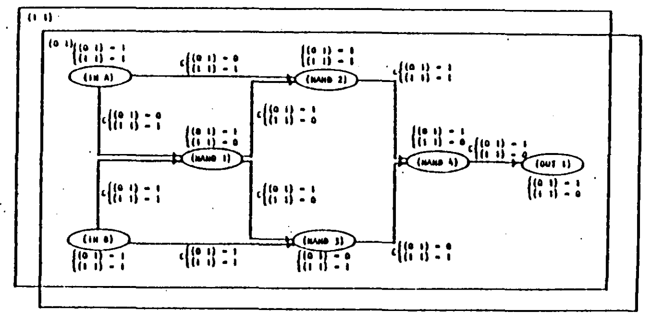
- $s \notin S$
- $((n\ s)\ v') \notin NSV$ for all $v' \in V$

[1] See alternative form on page 42.

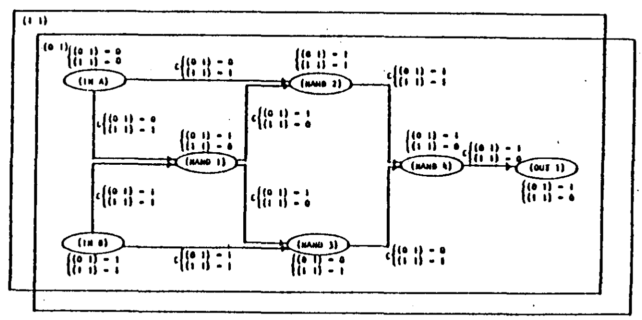
Illustrations



```
?(BUN '(IN A) 1 '(0 1))
(IN A)
```



```
?(BUN '(IN A) 0 '(1 1))
(IN A)
```



```
?(BUN '(IN A) 0 '(0 1))
(IN A)

?(BUN '(IN X) 0 '(0 1))
*** BUN ERROR: (IN X) IS NOT A NODE IN SPACE (0 1)

?(BUN '(IN A) 0 'SX)
*** BUN ERROR: SX IS NOT A SPACE
```

(BUS *space value*) Bind Unqualified Space \'bəs\

Informal Definition

The pseudo-function BUS is an EXPR which has the effect of binding *space* to *value*. Given *space* and *value*, BUS binds *space* to *value*. If *space* is already bound to *value*, BUS has no effect. BUS returns *space*.

space = *value*

error condition:

- *space* does not exist

Formal Definition

BUS[s,v] = s

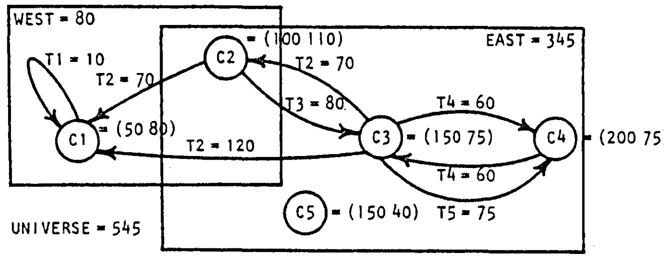
with effects:

$$SV := (SV - \{(s\ v') \mid v' \in V\}) \cup \{(s\ v)\}$$

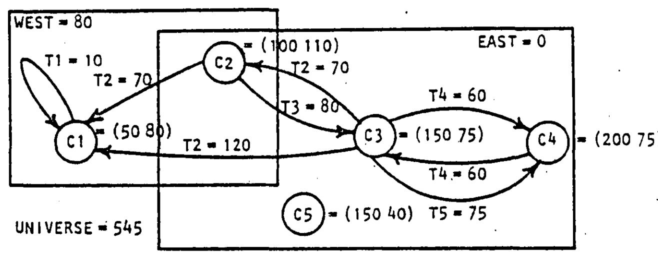
error condition:

- $s \notin S$

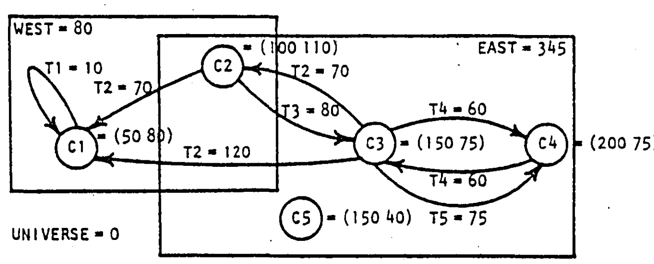
Illustrations



?(BUS 'EAST 0)
EAST



?(BUS 'UNIVERSE 0)
UNIVERSE



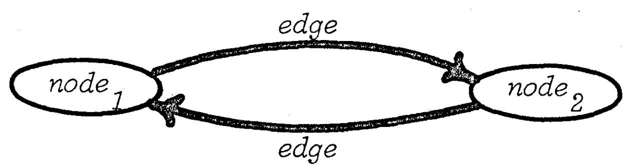
?(BUS 'EAST 345)
EAST

?(BUS 'SX 0)
*** BUS ERROR: SX IS NOT A SPACE

(CAP $node_1$ *edge* $node_2$)[1] Create Adjacent Pairs \'kap\

Informal Definition

The pseudo-function CAP is an EXPR which has the effect of creating the adjacent pairs (*edge* $node_2$) of $node_1$ in the universal space. Given $node_1$, *edge*, and $node_2$, CAP creates *edge* in UNIVERSE from $node_1$ to $node_2$ bound to NIL and *edge* from $node_2$ to $node_1$ bound to NIL. CAP has no effect on edges which already exist. Therefore, CAP only has an effect if either the outpointing or inpointing pair does not already exist. CAP returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

CAP[n,g,m] = n

with effects:

COP[n,g,m]

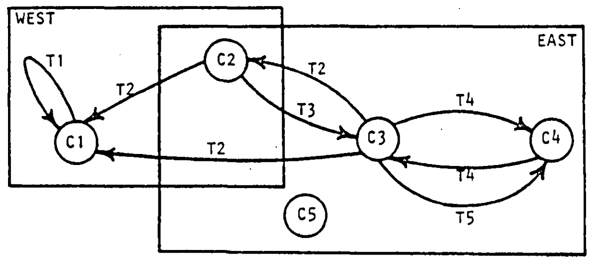
CIP[n,g,m]

error conditions:

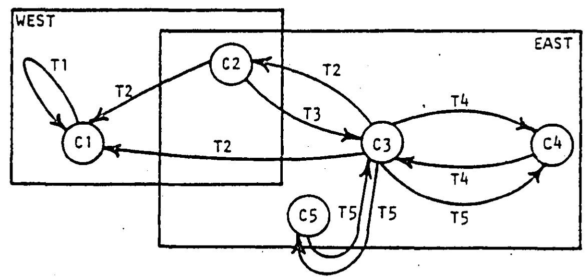
- $n \notin N$
- $m \notin N$

[1]See alternative form on page 50.

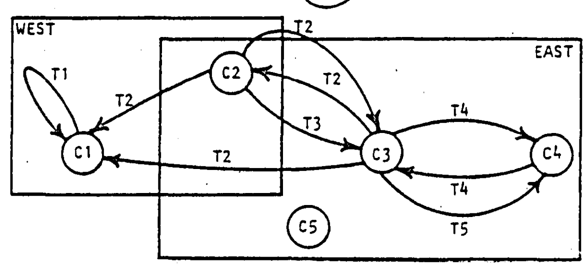
Illustrations



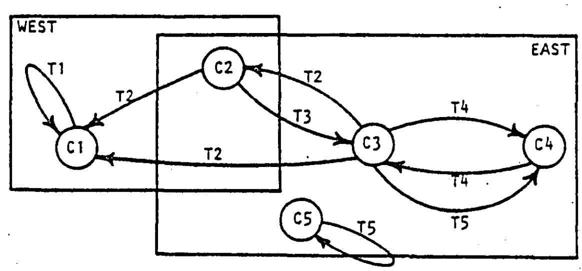

```
?(CAP 'C5 'T5 'C3)
C5
```




```
?(CAP 'C2 'T2 'C3)
C2
```




```
?(CAP 'C5 'T5 'C5)
C5
```

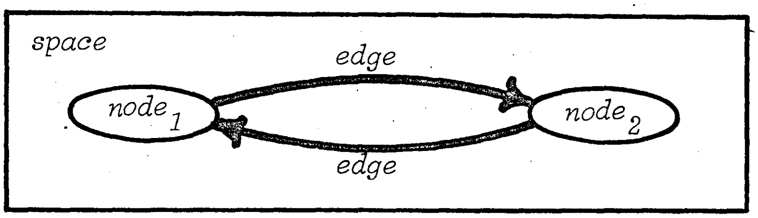



```
?(CAP 'C3 'T4 'C4)
C3

?(CAP 'C1 'T1 'C1)
C1

?(CAP 'CX 'T5 'C3)
*** CAP ERROR: CX IS NOT A NODE

?(CAP 'C5 'T5 'CX)
*** CAP ERROR: CX IS NOT A NODE
```

(CAP *node*$_1$ *edge* *node*$_2$ *space*)[1] Create Adjacent Pairs \'kap\

Informal Definition

The pseudo-function CAP is an EXPR which has the effect of creating the adjacent pairs (*edge* *node*$_2$) of *node*$_1$ in *space*. Given *node*$_1$, *edge*, *node*$_2$, and *space*, CAP creates *edge* in *space* from *node*$_1$ to *node*$_2$ bound to NIL and *edge* from *node*$_2$ to *node*$_1$ bound to NIL. If either of the edges did not already exist in the universal space, CAP also adds the missing ones to UNIVERSE bound to NIL. CAP has no effect on edges which already exist in *space*. Therefore, CAP only has an effect if either the outpointing or inpointing pair does not already exist in *space*. CAP returns *node*$_1$.



error conditions:

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

Formal Definition

CAP[n,g,m,s] = n

with effects:

COP[n,g,m,s]

CIP[n,g,m,s]

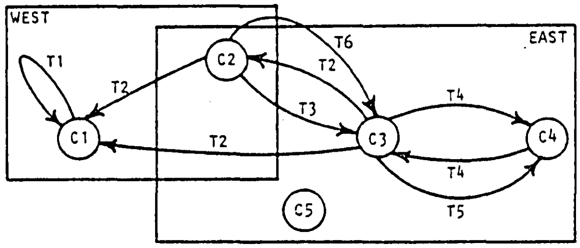
error conditions:

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

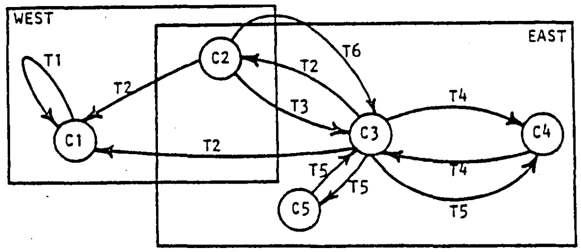
[1] See alternative form on page 48.

CAP

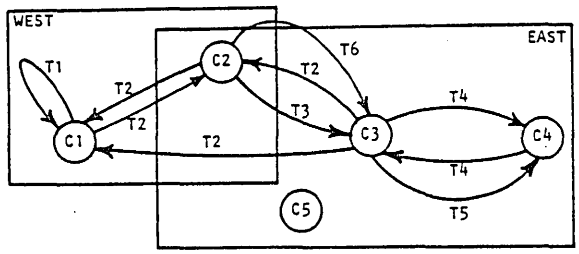
Illustrations



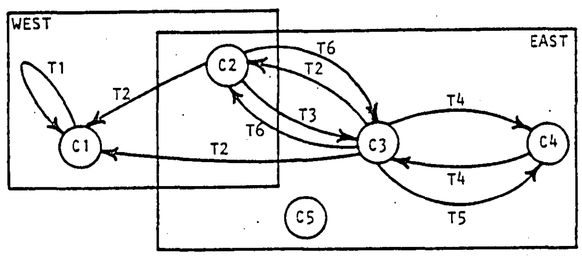

```
?(CAP 'C5 'T5 'C3 'EAST)
C5
```



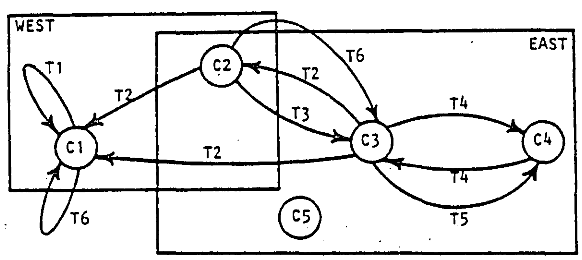

```
?(CAP 'C1 'T2 'C2 'WEST)
C1
```




```
?(CAP 'C2 'T6 'C3 'EAST)
C2
```




```
?(CAP 'C1 'T6 'C1 'UNIVERSE)
C1
```




```
?(CAP 'C3 'T4 'C4 'EAST)
C3

?(CAP 'C5 'T4 'C3 'WEST)
*** CAP ERROR: C5 IS NOT A NODE IN SPACE WEST

?(CAP 'CX 'T5 'C3 'EAST)
*** CAP ERROR: CX IS NOT A NODE IN SPACE EAST

?(CAP 'C5 'T5 'CX 'EAST)
*** CAP ERROR: CX IS NOT A NODE IN SPACE EAST
```

(CIP *node*$_1$ *edge* *node*$_2$)[1] Create Inpointing Pair \'kip\

Informal Definition

The pseudo-function CIP is an EXPR which has the effect of creating the inpointing pair (*edge* *node*$_2$) from *node*$_1$ in the universal space. Given *node*$_1$, *edge*, and *node*$_2$, CIP creates *edge* in UNIVERSE from *node*$_2$ to *node*$_1$ bound to NIL. If the pair already exists, CIP has no effect. CIP returns *node*$_1$.

error conditions:

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

Formal Definition

CIP[n,g,m] = n

with effects:

if (m g n) $\notin$ NGN

then NGN := NGN $\cup$ {(m g n)}

NGNSV := NGNSV $\cup$ {(((m g n) UNIVERSE) NIL)}

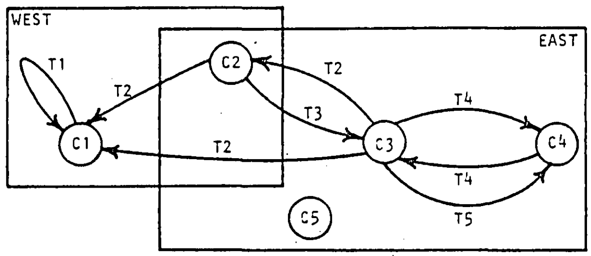
error conditions:

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

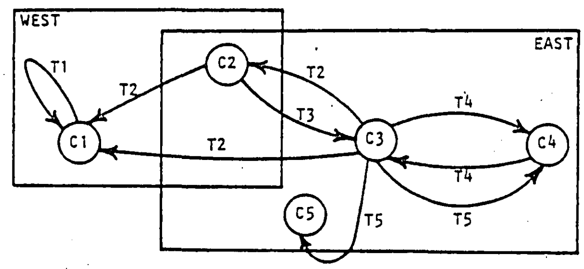
---

[1] See alternative form on page 54.

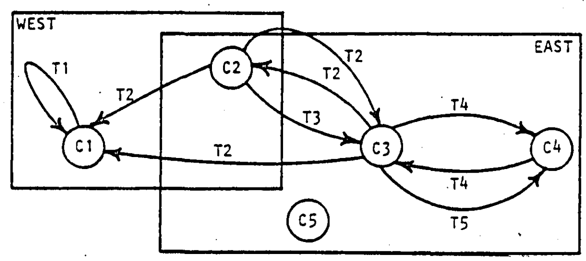
Illustrations



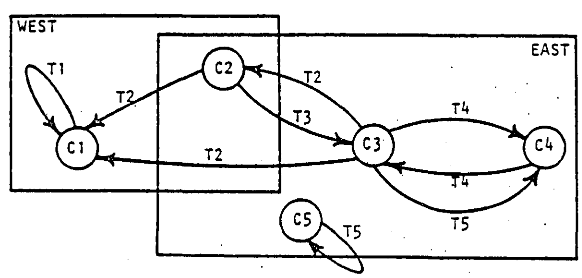
?(CIP 'C5 'T5 'C3)
C5



?(CIP 'C3 'T2 'C2)
C3



?(CIP 'C5 'T5 'C5)
C5



?(CIP 'C2 'T2 'C3)
C2

?(CIP 'C1 'T1 'C1)
C1

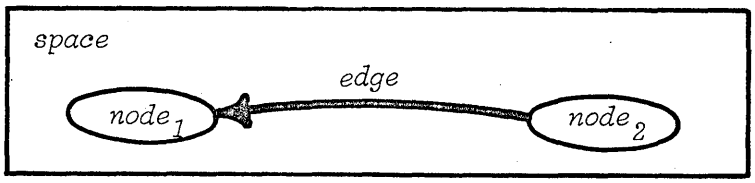
?(CIP 'CX 'T5 'C3)
*** CIP ERROR: CX IS NOT A NODE

?(CIP 'C5 'T5 'CX)
*** CIP ERROR: CX IS NOT A NODE

(CIP *node*$_1$ *edge* *node*$_2$ *space*)[1] Create Inpointing Pair \'kip\

Informal Definition

The pseudo-function CIP is an EXPR which has the effect of creating the inpointing pair (*edge* *node*$_2$) of *node*$_1$ in *space*. Given *node*$_1$, *edge*, *node*$_2$, and *space*, CIP creates *edge* from *node*$_2$ to *node*$_1$ in *space* and binds the edge to NIL in *space*. If the edge did not already exist in the universal space, CIP also adds it to UNIVERSE bound to NIL. If the pair already exists in *space*, CIP has no effect. CIP returns *node*$_1$.



error conditions:

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

Formal Definition

CIP[n,g,m,s] = n

with effects:

CIP[n,g,m]

If (((m g n) s) v) $\notin$ NGNSV for all v $\in$ V

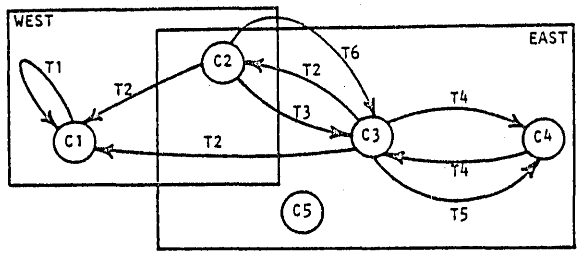
then NGNSV := NGNSV $\cup$ {(((m g n) s) NIL)}

error conditions:

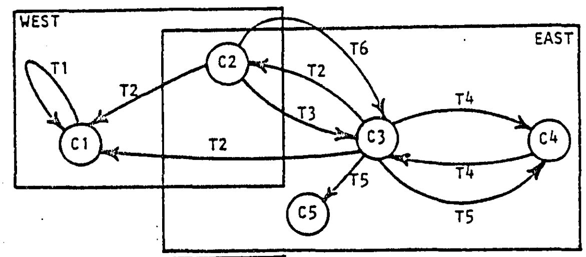
- ((n s) v) $\notin$ NSV for all v $\in$ V
- ((m s) v) $\notin$ NSV for all v $\in$ V
- s $\notin$ S

[1]See alternative form on page 52.

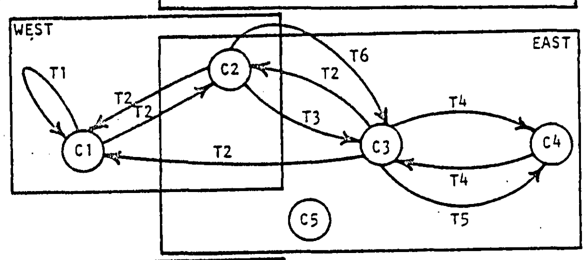
Illustrations



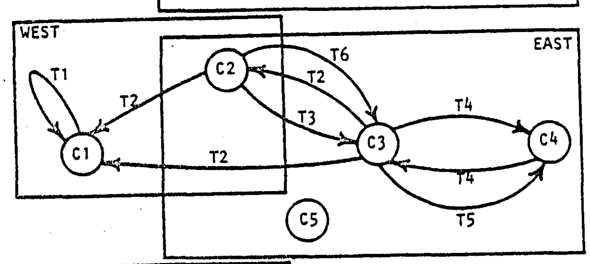
?(CIP 'C5 'T5 'C3 'EAST)
C5



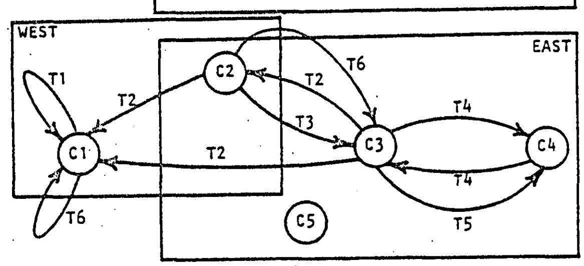
?(CIP 'C2 'T2 'C1 'WEST)
C2



?(CIP 'C3 'T6 'C2 'EAST)
C3



?(CIP 'C1 'T6 'C1 'UNIVERSE)
C1



?(CIP 'C3 'T4 'C4 'EAST)
C3

?(CIP 'C5 'T5 'C3 'WEST)
*** CIP ERROR: C3 IS NOT A NODE IN SPACE WEST

?(CIP 'CX 'T5 'C3 'EAST)
*** CIP ERROR: CX IS NOT A NODE IN SPACE EAST

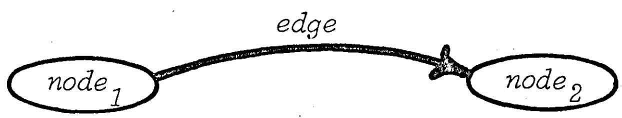
?(CIP 'C5 'T5 'CX 'EAST)
*** CIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(CIP 'C5 'T5 'C3 'SX)
*** CIP ERROR: SX IS NOT A SPACE

(COP *node*$_1$ *edge* *node*$_2$)[1] Create Outpointing Pair \'kä̈p\

Informal Definition

The pseudo-function COP is an EXPR which has the effect of creating the outpointing pair (*edge* *node*$_2$) from *node*$_1$ in the universal space. Given *node*$_1$, *edge*, and *node*$_2$, COP creates *edge* in UNIVERSE from *node*$_1$ to *node*$_2$ bound to NIL. If the pair already exists, COP has no effect. COP returns *node*$_1$.



error conditions:

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

Formal Definition

COP[n,g,m] = n

with effects:

if (n g m) $\notin$ NGN

then NGN := NGN $\cup$ {(n g m)}

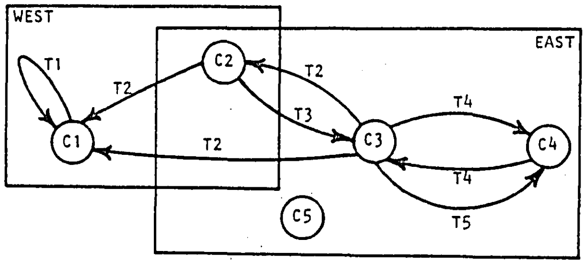
NGNSV := NGNSV $\cup$ {(((n g m) UNIVERSE) NIL)}

error conditions:

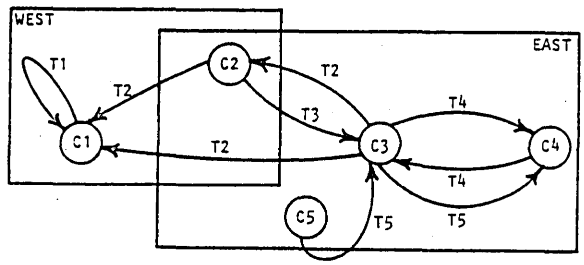
- n $\notin$ N
- m $\notin$ N

[1]See alternative form on page 58.

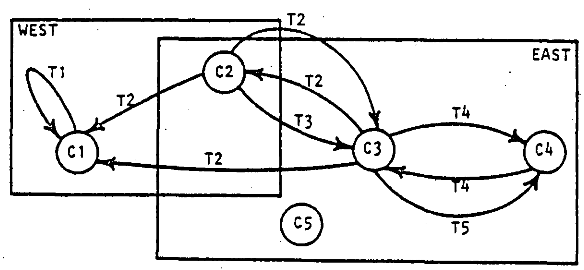
Illustrations



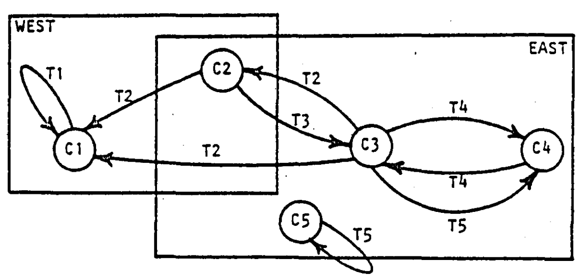

```
?(COP 'C5 'T5 'C3)
C5
```




```
?(COP 'C2 'T2 'C3)
C2
```




```
?(COP 'C5 'T5 'C5)
C5
```

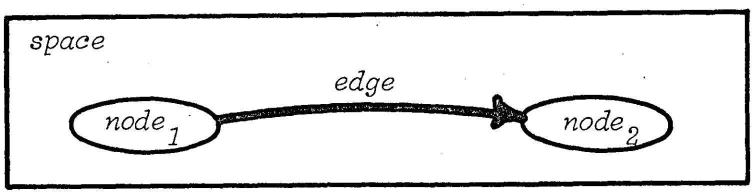



```
?(COP 'C3 'T2 'C2)
C3

?(COP 'C1 'T1 'C1)
C1

?(COP 'CX 'T5 'C3)
*** COP ERROR: CX IS NOT A NODE

?(COP 'C5 'T5 'CX)
*** COP ERROR: CX IS NOT A NODE
```

(COP *node*$_1$ *edge* *node*$_2$ *space*)[1] <u>C</u>reate <u>O</u>utpointing <u>P</u>air \'käp\

<u>Informal Definition</u>

The pseudo-function COP is an EXPR which has the effect of creating the outpointing pair (*edge* *node*$_2$) from *node*$_1$ in *space*. Given *node*$_1$, *edge*, *node*$_2$, and *space*, COP creates *edge* from *node*$_1$ to *node*$_2$ in *space* and binds the edge to NIL in *space*. If the edge did not already exist in the universal space, COP also adds it to UNIVERSE bound to NIL. If the pair already exists in *space*, COP has no effect. COP returns *node*$_1$.



error conditions:

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

<u>Formal Definition</u>

COP[n,g,m,s] = n

with effects:

COP[n,g,m]

If $(((n\ g\ m)\ s)\ v) \notin$ NGNSV for all $v \in V$

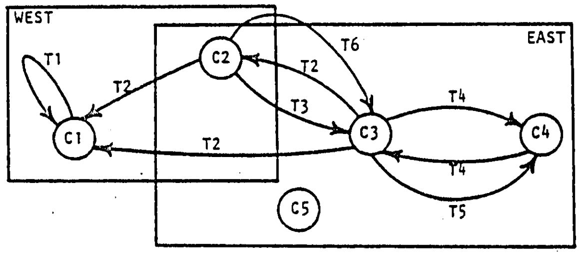
then NGNSV := NGNSV $\cup$ {(((n g m) s) NIL)}

error conditions:

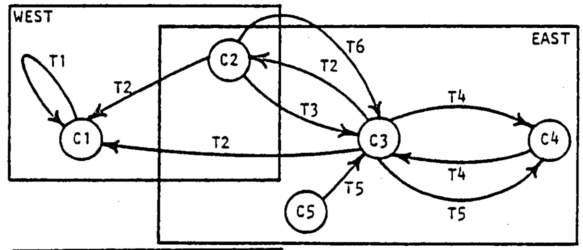
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $((m\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1] See alternative form on page 56.

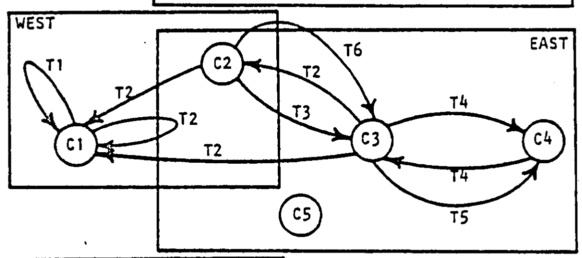
Illustrations



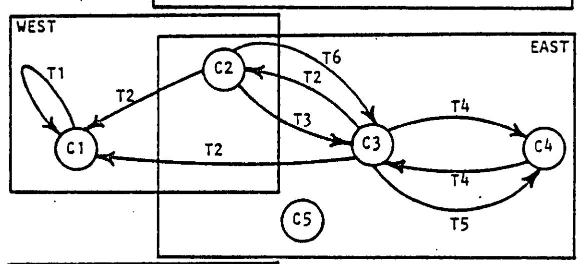

```
?(COP 'C5 'T5 'C3 'EAST)
C5
```



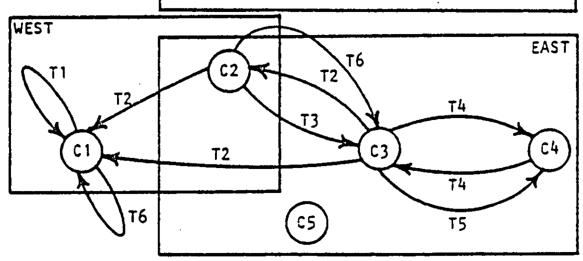

```
?(COP 'C1 'T2 'C1 'WEST)
C1
```




```
?(COP 'C2 'T6 'C3 'EAST)
C2
```




```
?(COP 'C1 'T6 'C1 'UNIVERSE)
C1
```




```
?(COP 'C3 'T4 'C4 'EAST)
C3

?(COP 'C5 'T5 'C3 'WEST)
*** COP ERROR: C5 IS NOT A NODE IN SPACE WEST

?(COP 'CX 'T5 'C3 'EAST)
*** COP ERROR: CX IS NOT A NODE IN SPACE EAST

?(COP 'C5 'T5 'CX 'EAST)
*** COP ERROR: CX IS NOT A NODE IN SPACE EAST

?(COP 'C5 'T5 'C3 'SX)
*** COP ERROR: SX IS NOT A SPACE
```

(CUN *node*)[1] Create Unqualified Node \\'kən\\

Informal Definition

The pseudo-function CUN is an EXPR which has the effect of creating *node* in the universal space. Given *node*, CUN creates *node* in UNIVERSE bound to NIL. If *node* already exists, CUN has no effect. CUN returns *node*.

Formal Definition

CUN[n] = n

with effects:

if $n \notin N$

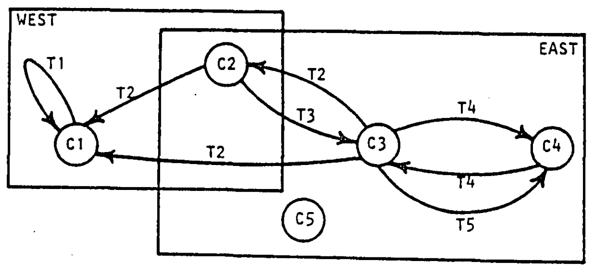
then $N := N \cup \{n\}$

$NSV := NSV \cup \{((n\ \mathrm{UNIVERSE})\ \mathrm{NIL})\}$

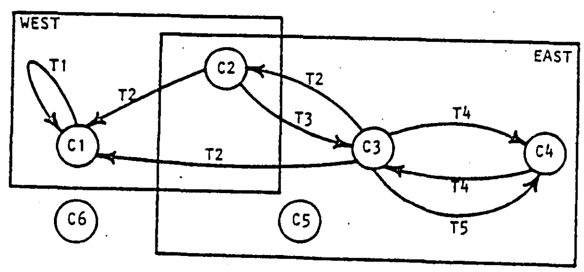
---

[1]See alternative form on page 62.

Illustrations




```
?(CUN 'C6)
C6
```

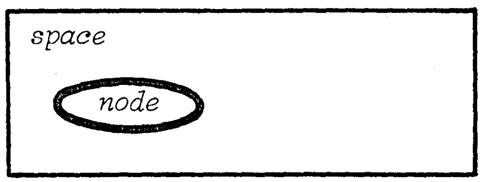



```
?(CUN 'C5)
C5
```

(CUN *node space*)[1] Create Unqualified Node \\'kən\\

Informal Definition

The pseudo-function CUN is an EXPR which has the effect of creating *node* in *space*. Given *node* and *space*, CUN adds *node* to *space* bound to NIL. If *node* did not already exist, CUN also adds it to UNIVERSE bound to NIL. CUN has no effect if *node* already exists in *space*. CUN returns *node*.



error condition:

- *space* does not exist

Formal Definition

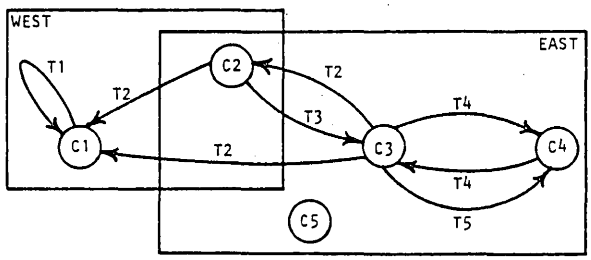
CUN[n,s] = n
with effects:
CUN[n]
if (n s) $\notin$ NS
then NSV := NSV $\cup$ {((n s) NIL)}

error condition:

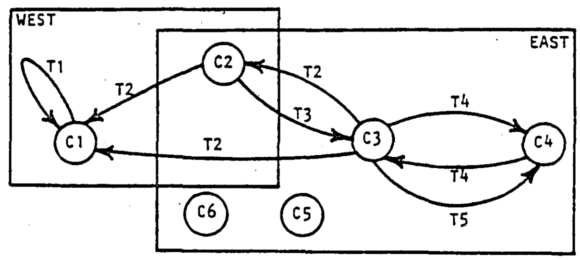
- s $\notin$ S

[1]See alternative form on page 60.

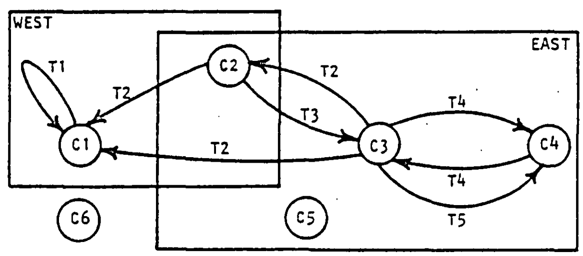
<u>Illustrations</u>



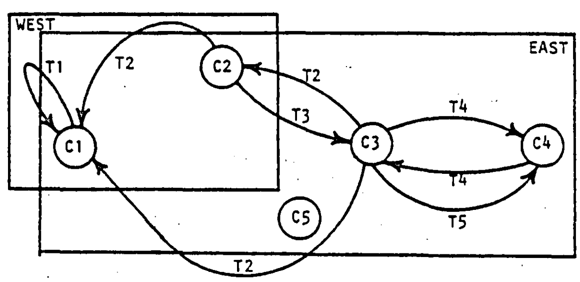

```
?(CUN 'C6 'EAST)
C6
```




```
?(CUN 'C6 'UNIVERSE)
C6
```




```
?(CUN 'C1 'EAST)
C1
```




```
?(CUN 'C5 'EAST)
C5

?(CUN 'C6 'SX)
*** CUN ERROR: SX IS NOT A SPACE
```

(CUS *space*) Create Unqualified Space \'kəs\

Informal Definition

The pseudo-function CUS is an EXPR which has the effect of creating *space*. Given *space*, CUS creates *space* bound to NIL. CUS has no effect if *space* already exists. CUS returns *space*.

Formal Definition

CUS[s] = s

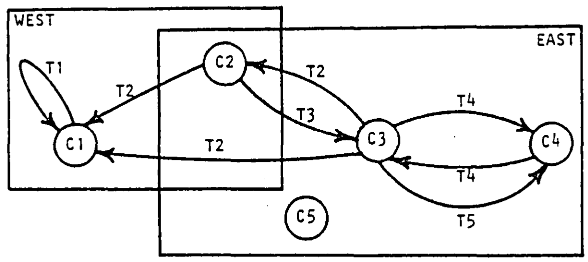
with effects:

if $s \notin S$

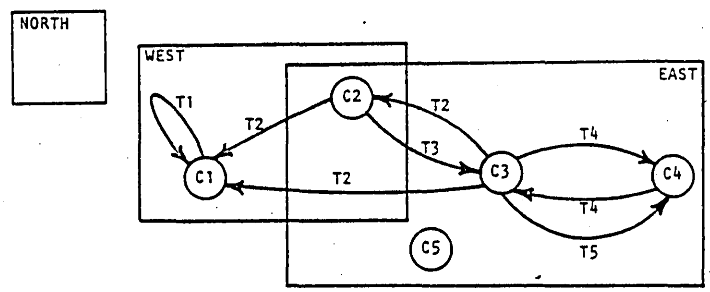
then $S := S \cup \{s\}$

$SV := SV \cup \{(s\ \mathrm{NIL})\}$

Illustrations




```
?(CUS 'NORTH)
NORTH
```

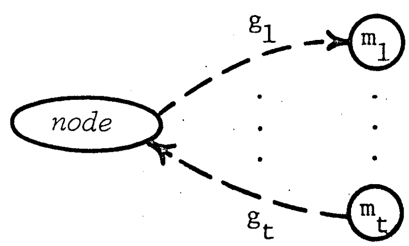



```
?(CUS 'EAST)
EAST

?(CUS 'UNIVERSE)
UNIVERSE
```

(DAG *node*)[1] Destroy Adjacent edGes \'dag\

Informal Definition

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



error condition:

- *node* does not exist

Formal Definition

DAG[n] = n

with effects:

DOG[n]

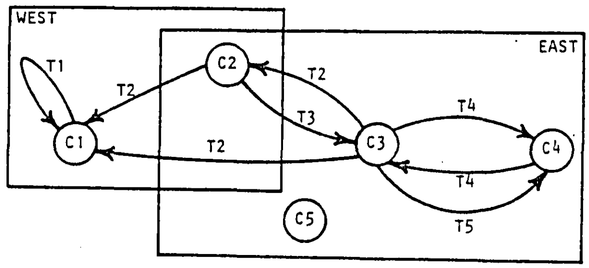
DIG[n]

error condition:

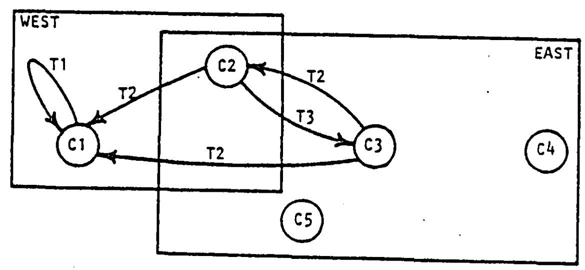
- $n \notin N$

[1]See alternative form on page 68.

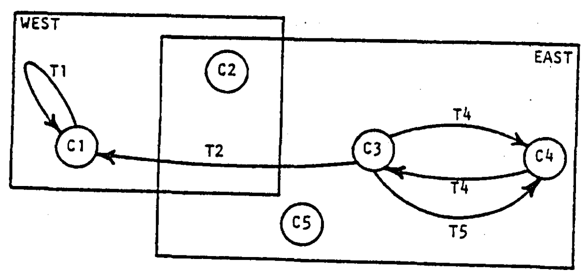
Illustrations



?(DAG 'C4)
C4



?(DAG 'C2)
C2



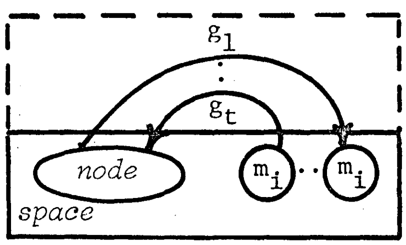
?(DAG 'C5)
C5

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

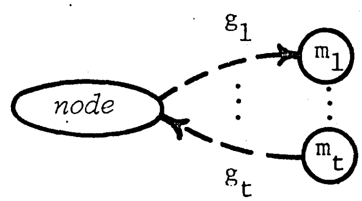
(DAG *node space*)[1] Destroy Adjacent edGes \'dag\

Informal Definition

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



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DAG[n,s] = n

with effects:

DOG[n,s]

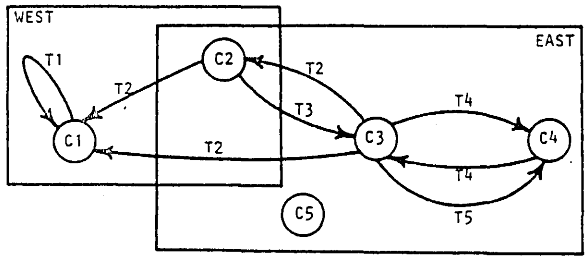
DIG[n,s]

error conditions:

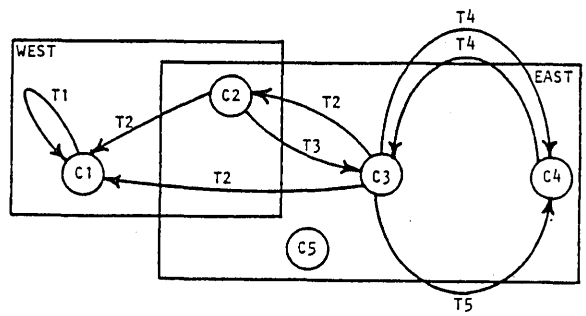
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 66.

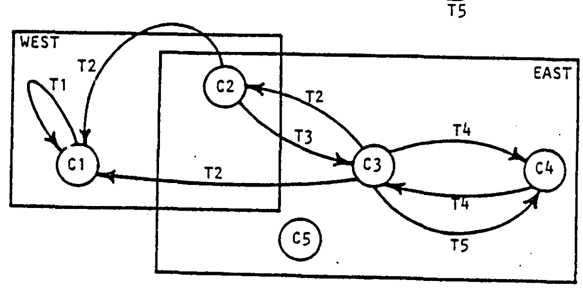
Illustrations



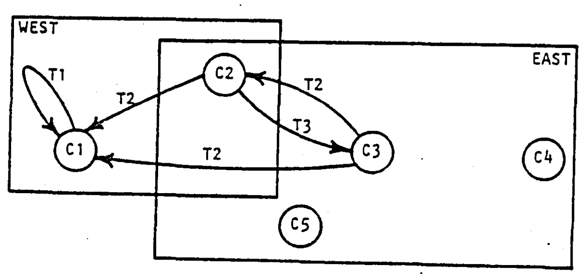
?(DAG 'C4 'EAST)
C4



?(DAG 'C2 'WEST)
C2



?(DAG 'C4 'UNIVERSE)
C4



?(DAG 'C5 'EAST)
C5

?(DAG 'CX 'EAST)

*** DAG ERROR: CX IS NOT A NODE IN SPACE EAST

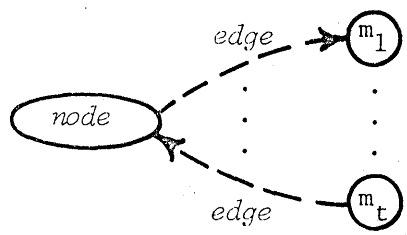
?(DAG 'C4 'SX)

*** DAG ERROR: SX IS NOT A SPACE

(DAGG *node edge*)[1] Destroy Adjacent edGes given an edGe \'dag-gə\

Informal Definition

The pseudo-function DAGG is an EXPR which has the effect of destroying all adjacent instances of *edge* to and from *node*. Given *node* and *edge*, DAGG destroys all instances of *edge* which point to or from *node*. If no such edges exist, DAGG has no effect. DAGG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DAGG[n,g] = n

with effects:

DOGG[n,g]

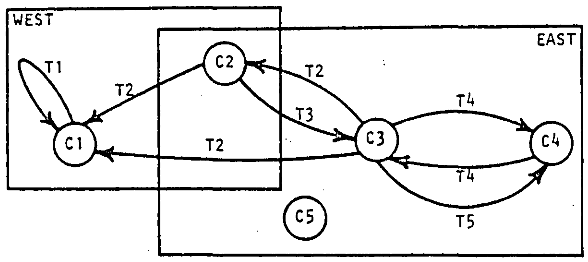
DIGG[n,g]

error condition:

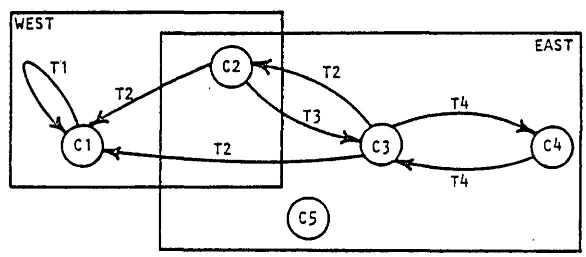
- $n \notin N$

[1]See alternative form on page 72.

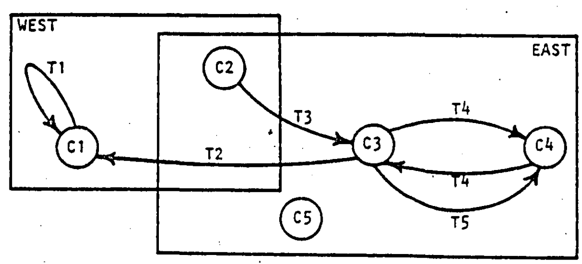
Illustrations




```
?(DAGG 'C4 'T5)
C4
```




```
?(DAGG 'C2 'T2)
C2
```

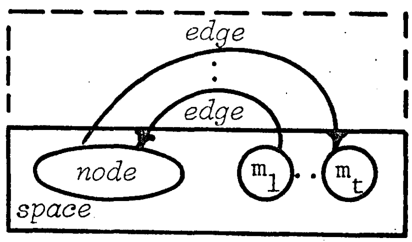



```
?(DAGG 'C5 'TX)
C5

?(DAGG 'CX 'T5)
*** DAGG ERROR: CX IS NOT A NODE
```

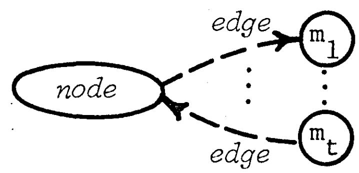
(DAGG *node edge space*)[1] Destroy Adjacent edGes given an edGe \'dag-gə\

Informal Definition

The pseudo-function DAGG is an EXPR which has the effect of destroying all adjacent instances of *edge* to and from *node* in *space*. Given *node*, *edge*, and *space*, DAGG removes all instances of *edge* from *space* which point to or from *node*. If *space* is UNIVERSE, DAGG removes each such *edge* from all spaces. If no such *edge* exists, DAGG has no effect. DAGG returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

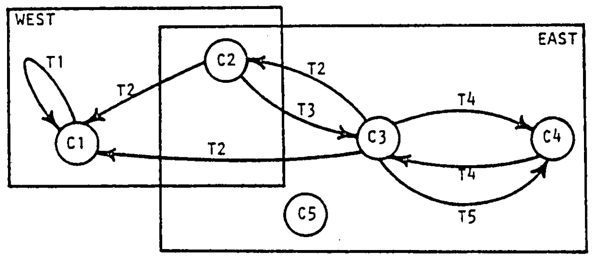
DAGG[n,g,s] = n
with effects:
DOGG[n,g,s]
DIGG[n,g,s]

error conditions:

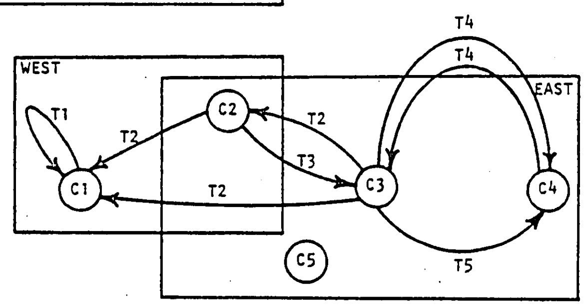
- $((n\ s)\ v) \notin NSV$ for all $v \in V$
- $s \notin S$

[1] See alternative form on page 70.

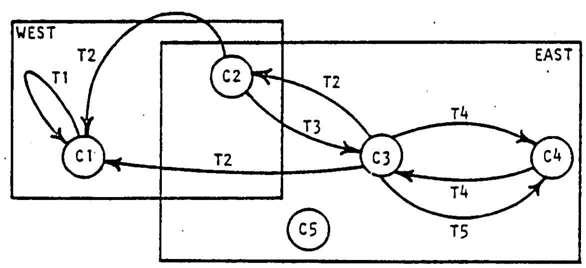
Illustrations



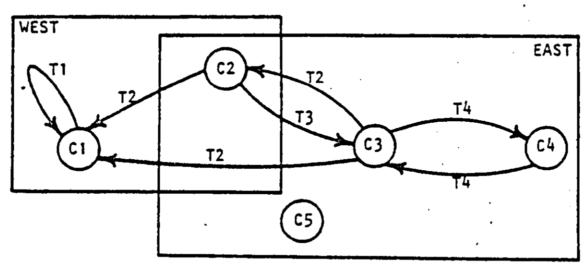

```
?(DAGG 'C4 'T4 'EAST)
C4
```




```
?(DAGG 'C2 'T2 'WEST)
C2
```




```
?(DAGG 'C4 'T5 'UNIVERSE)
C4
```

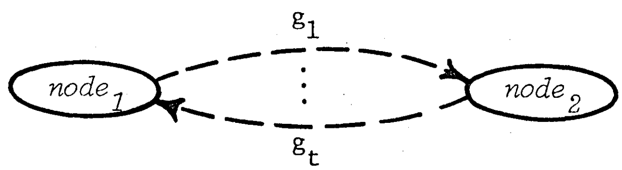



```
?(DAGG 'C3 'TX 'EAST)
C3

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

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

(DAGN $node_1$ $node_2$)[1] Destroy Adjacent edGes given a Node \'dag-in\

Informal Definition

The pseudo-function DAGN is an EXPR which has the effect of destroying all adjacent edges of $node_1$ connected to $node_2$. Given $node_1$ and $node_2$, DAGN destroys all edges g_i where for each i, g_i either points from $node_1$ to $node_2$ or from $node_2$ to $node_1$. If no such edges exist, DAGN has no effect. DAGN returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

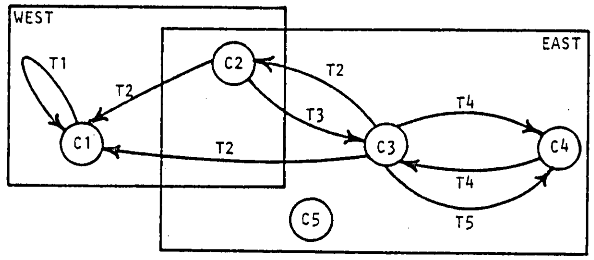
DAGN[n,m] = n
with effects:
DOGN[n,m]
DAGN[n,m]

error conditions:

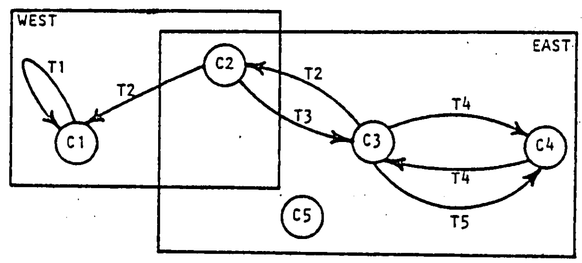
- $n \notin N$
- $m \notin N$

[1]See alternative form on page 76.

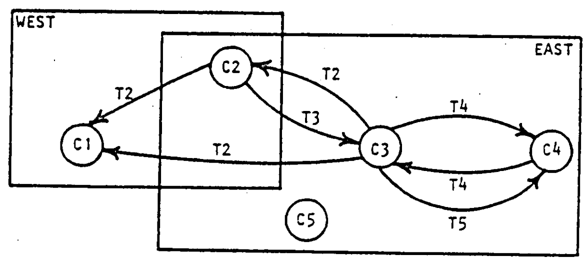
Illustrations



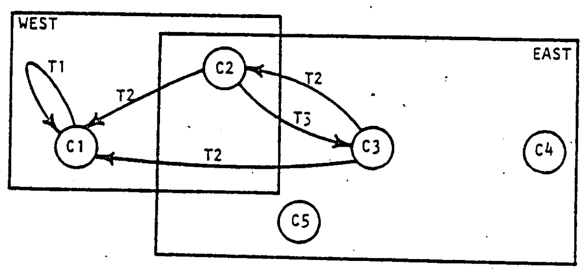
?(DAGN 'C1 'C3)
C1



?(DAGN 'C1 'C1)
C1



?(DAGN 'C3 'C4)
C3



?(DAGN 'C5 'C3)
C5

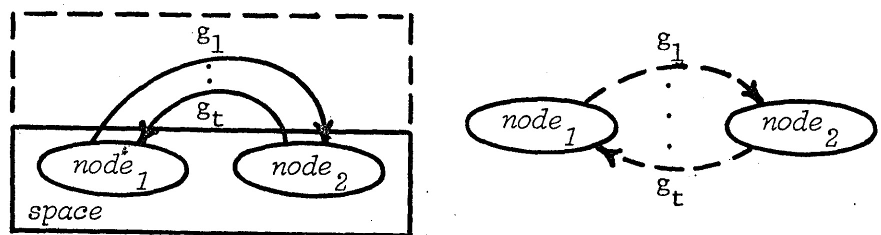
?(DAGN 'CX 'C3)
*** DAGN ERROR: CX IS NOT A NODE

?(DAGN 'C1 'CX)
*** DAGN ERROR: CX IS NOT A NODE

(DAGN $node_1$ $node_2$ *space*)[1] Destroy Adjacent edGes given a Node \'dag-in\

Informal Definition

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

DAGN[n,m,s] = n

with effects:

DOGN[n,m,s]

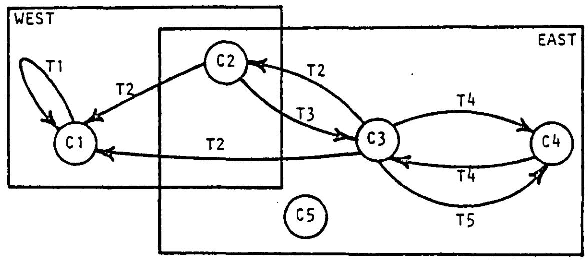
DIGN[n,m,s]

error conditions:

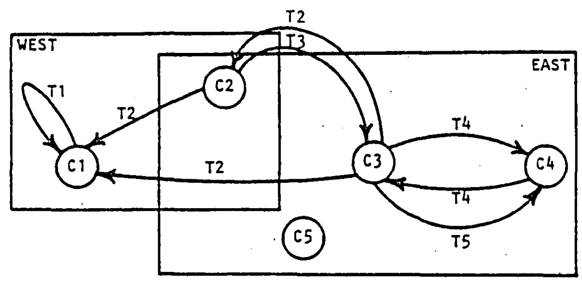
- ((n s) v) $\notin$ NSV for all v $\in$ V
- ((m s) v) $\notin$ NSV for all v $\in$ V
- s $\notin$ S

[1]See alternative form on page 74.

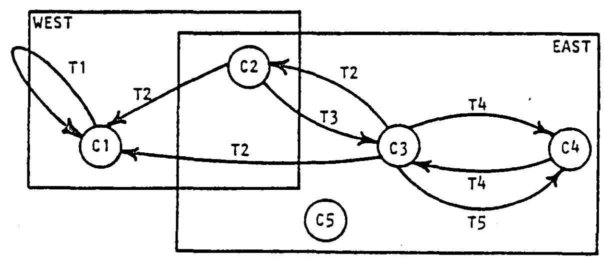
Illustrations



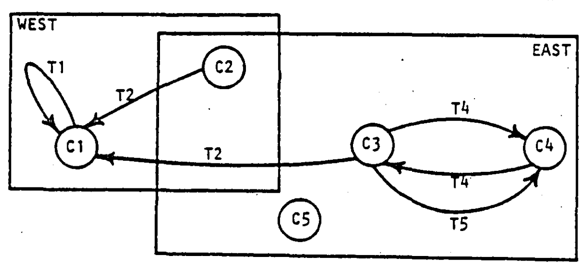
?(DAGN 'C2 'C3 'EAST)
C2



?(DAGN 'C1 'C1 'WEST)
C1



?(DAGN 'C2 'C3 'UNIVERSE)
C2



?(DAGN 'C5 'C3 'EAST)
C5

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

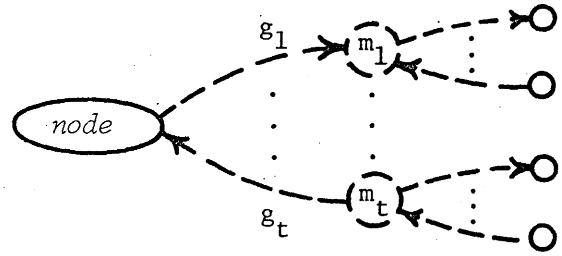
?(DAGN 'C5 'CX 'EAST)
*** DAGN ERROR: CX IS NOT A NODE IN SPACE EAST

?(DAGN 'C5 'C3 'SX)
*** DAGN ERROR: SX IS NOT A SPACE

(DAN *node*)[1] Destroy Adjacent Nodes \'dan\

Informal Definition

The pseudo-function DAN is an EXPR which has the effect of destroying all adjacent nodes of *node*. Given *node*, DAN destroys all nodes m_i where for each i, some edge g_i points to node m_i from *node* or points to *node* from node m_i. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DAN has no effect. DAN returns *node*.



error condition:

- *node* does not exist

Formal Definition

DAN[n] = n

with effects:

DON[n]

DIN[n]

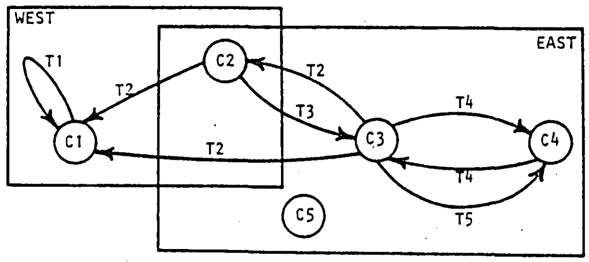
error condition:

- $n \notin N$

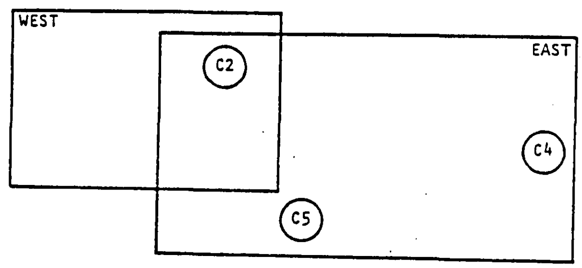
[1] See alternative form on page 80.

[2] See DAG on page 66.

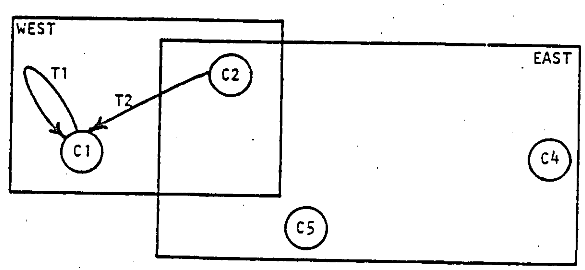
Illustrations



?(DAN 'C2)
C2



?(DAN 'C4)
C4



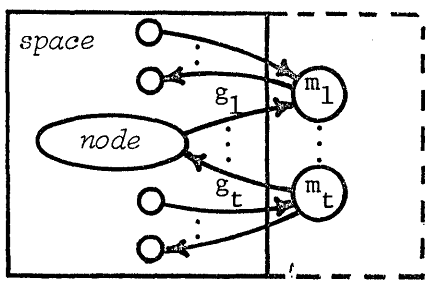
?(DAN 'C5)
C5

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

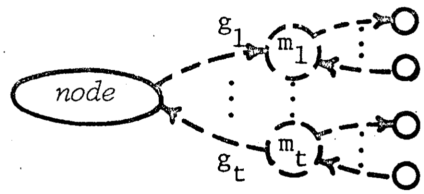
(DAN *node space*)[1] Destroy Adjacent Nodes \'dan\

Informal Definition

The pseudo-function DAN is an EXPR which has the effect of destroying all adjacent nodes of *node* in *space*. Given *node* and *space*, DAN removes all nodes m_i from *space* where for each i, some edge g_i points to node m_i from *node* or points to *node* from node m_i in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DAN removes each node m_i and all its adjacent edges from all spaces. If no such nodes exist, DAN has no effect. DAN returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DAN[n,s] = n
  with effects:
    DON[n,s]
    DIN[n,s]

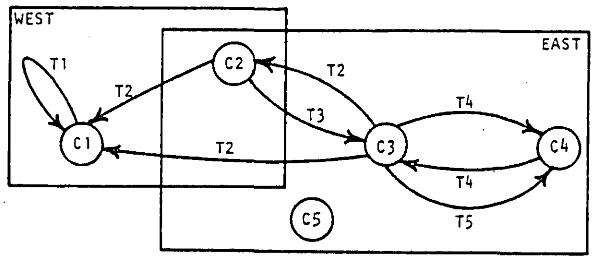
error conditions:

- $n \notin N$
- $s \notin S$

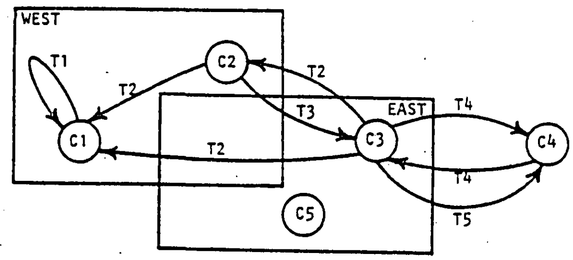
[1] See alternative form on page 78.

[2] See DAG on page 68.

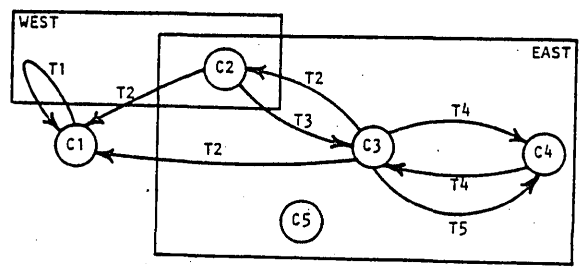
Illustrations



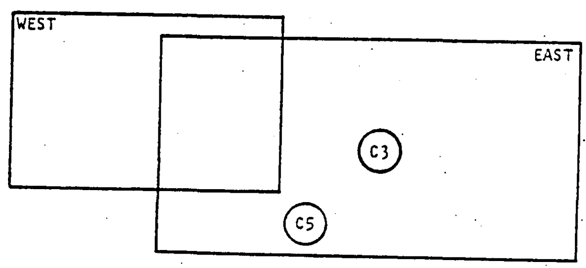
?(DAN 'C3 'EAST)
C3



?(DAN 'C2 'WEST)
C2



?(DAN 'C3 'UNIVERSE)
C3



?(DAN 'C5 'EAST)
C5

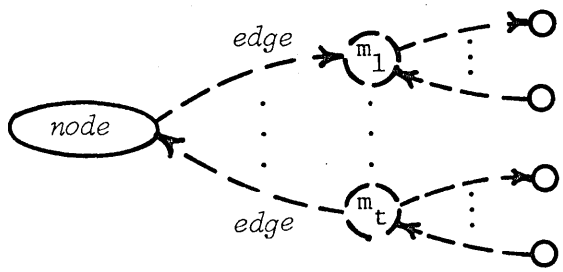
?(DAN 'CX 'EAST)
*** DAN ERROR: CX IS NOT A NODE IN SPACE EAST

?(DAN 'C3 'SX)
*** DAN ERROR: SX IS NOT A SPACE

(DANG *node edge*)[1] Destroy Adjacent Nodes given an edGe \'daŋ\

Informal Definition

The pseudo-function DANG is an EXPR which has the effect of destroying all adjacent nodes of *node* connected by *edge*. Given *node* and *edge*, DANG destroys all nodes m_i where for each i, *edge* points to node m_i from *node* or points to *node* from node m_i. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DANG has no effect. DANG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DANG[n,g] = n

with effects:

DONG[n,g]

DING[n,g]

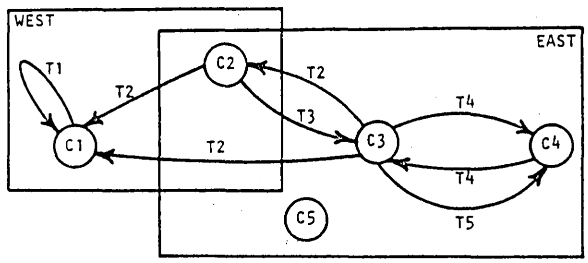
error condition:

- $n \notin N$

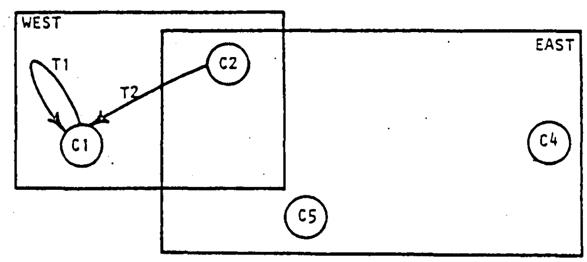
[1]See alternative form on page 84.

[2]See DAG on page 66.

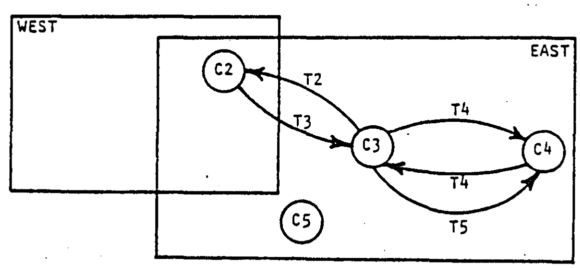
Illustrations




```
?(DANG 'C4 'T4)
C4
```




```
?(DANG 'C1 'T1)
C1
```

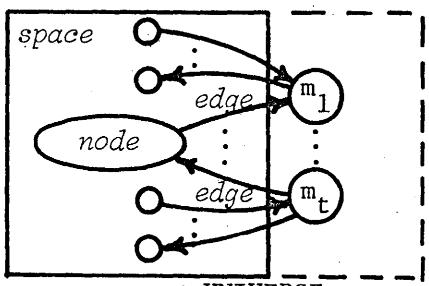



```
?(DANG 'C5 'TX)
C5

?(DANG 'CX 'T1)
*** DANG ERROR: CX IS NOT A NODE
```

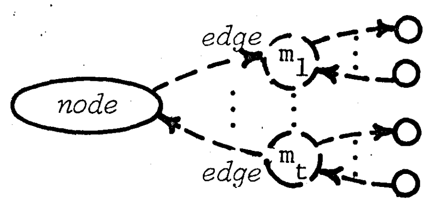
(DANG *node edge space*)[1] Destroy Adjacent Nodes given an edGe \'daŋ\

Informal Definition

The pseudo-function DANG is an EXPR which has the effect of destroying all adjacent nodes of *node* connected by *edge* in *space*. Given *node*, *edge*, and *space*, DANG removes all nodes m_i from *space* where for each i, *edge* points to node m_i from *node* or points to *node* from node m_i in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DANG removes each node m_i and all its adjacent edges from all spaces. If no such nodes exist, DANG has no effect. DANG returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DANG[n,g,s] = n

with effects:

DONG[n,g,s]

DING[n,g,s]

error conditions:

- $n \notin N$
- $s \notin S$

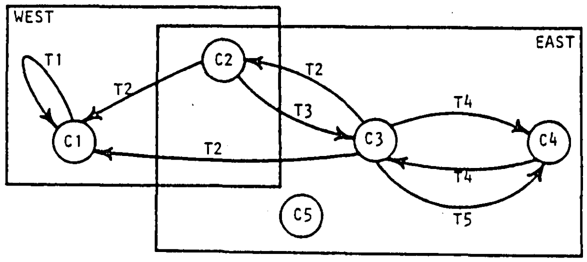
---

[1] See alternative form on page 82.

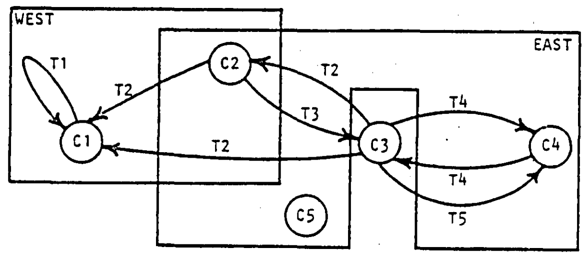
[2] See DAG on page 68.

)ANG

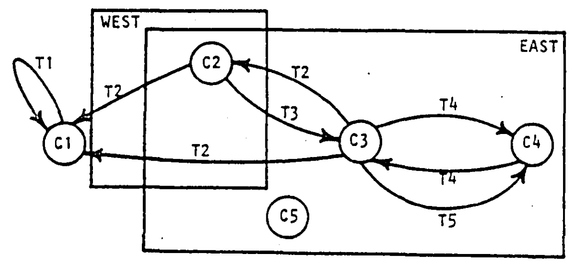
Illustrations



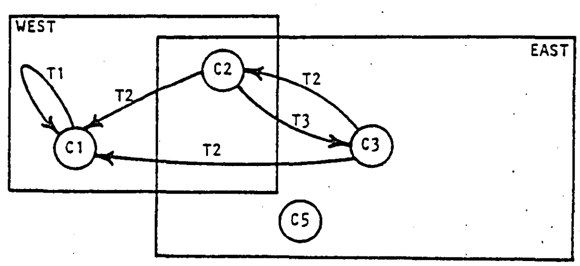

```
?(DANG 'C4 'T4 'EAST)
C4
```




```
?(DANG 'C1 'T1 'WEST)
C1
```




```
?(DANG 'C3 'T5 'UNIVERSE)
C3
```

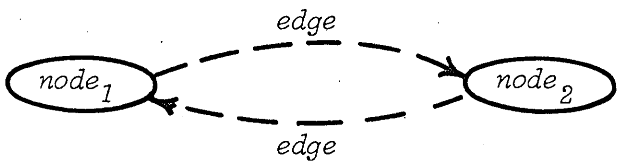



```
?(DANG 'C4 'TX 'EAST)
C4

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

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

(DAP *node*$_1$ *edge* *node*$_2$)[1] Destroy Adjacent Pairs \'dap\

Informal Definition

The pseudo-function DAP is an EXPR which has the effect of destroying the adjacent pairs (*edge* *node*$_2$) of *node*$_1$. Given *node*$_1$, *edge*, and *node*$_2$, DAP destroys *edge* from *node*$_1$ to *node*$_2$ and from *node*$_2$ to *node*$_1$. DAP has no effect if neither the outpointing nor inpointing pair exists. DAP returns *node*$_1$.



error conditions:

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

Formal Definition

DAP[n,g,m] = n

with effects:

DOP[n,g,m]

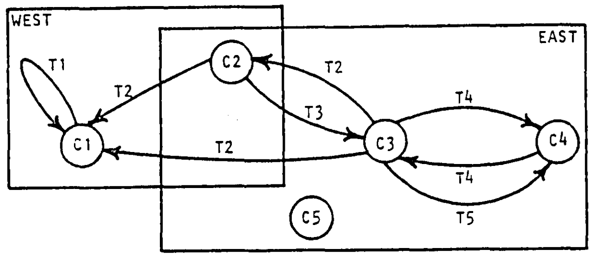
DIP[n,g,m]

error conditions:

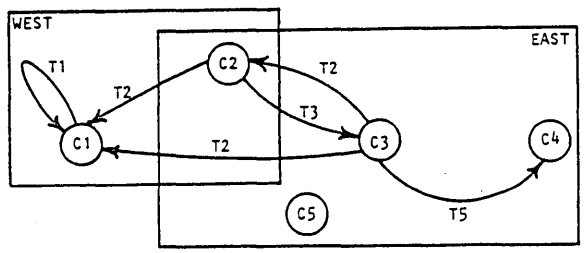
- $n \notin N$
- $m \notin N$

[1]See alternative form on page 88.

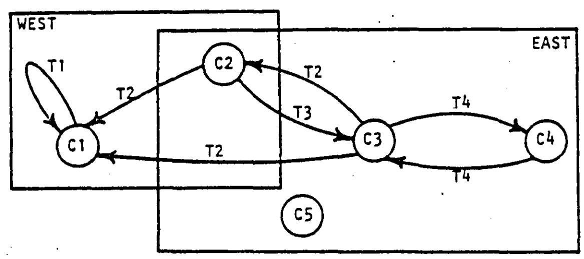
Illustrations



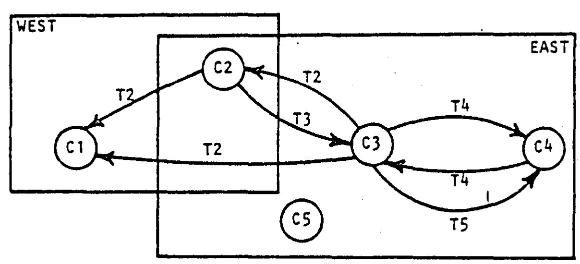
?(DAP 'C3 'T4 'C4)
C3



?(DAP 'C4 'T5 'C3)
C4



?(DAP 'C1 'T1 'C1)
C1



?(DAP 'C5 'TX 'C3)
C5

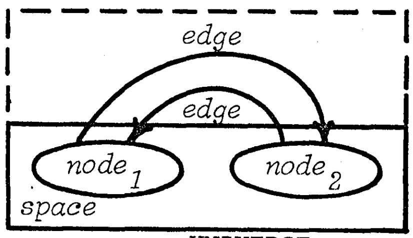
?(DAP 'CX 'T1 'C3)
*** DAP ERROR: CX IS NOT A NODE

?(DAP 'C5 'T1 'CX)
*** DAP ERROR: CX IS NOT A NODE

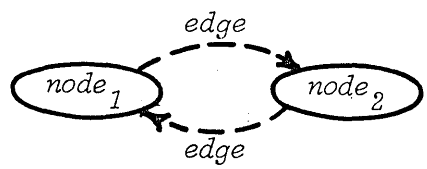
(DAP $node_1$ *edge* $node_2$ *space*)[1] Destroy Adjacent Pairs \'dap\

Informal Definition

The pseudo-function DAP is an EXPR which has the effect of destroying the adjacent pairs (*edge* $node_2$) of $node_1$ in *space*. Given $node_1$, *edge*, $node_2$, and *space*, DAP removes *edge* pointing from $node_1$ to $node_2$ from *space* and *edge* pointing from $node_2$ to $node_1$ from *space*. If *space* is UNIVERSE, DAP removes *edge* pointing from $node_1$ to $node_2$ and *edge* pointing from $node_2$ to $node_1$ from all spaces. DAP has no effect if neither the outpointing nor inpointing pair exists in *space*. DAP 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

DAP[n,g,m,s] = n

with effects:

DOP[n,g,m,s]

DIP[n,g,m,s]

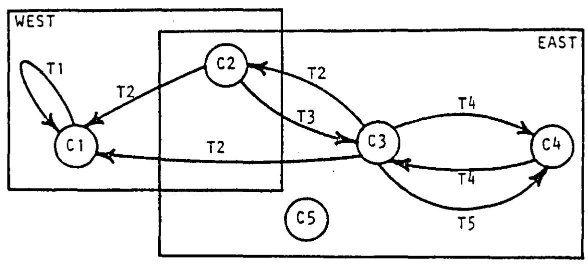
error conditions:

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

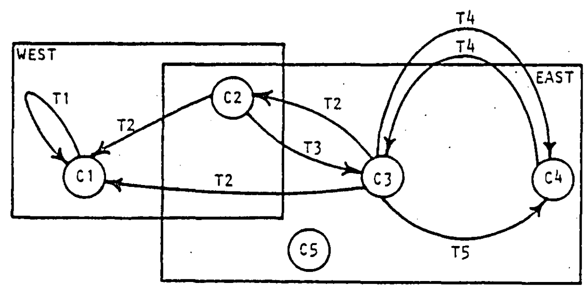
---

[1] See alternative form on page 86.

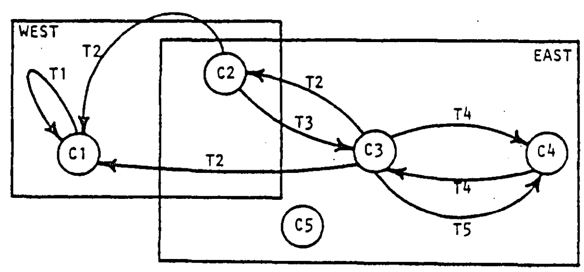
Illustrations



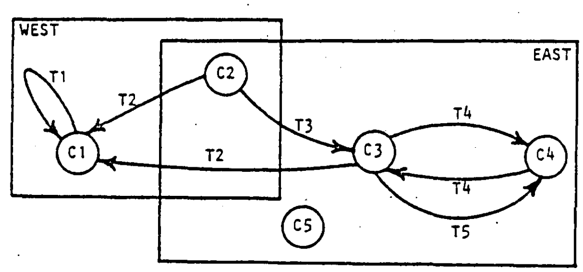

```
?(DAP 'C3 'T4 'C4 'EAST)
C3
```




```
?(DAP 'C2 'T2 'C1 'WEST)
C2
```




```
?(DAP 'C2 'T2 'C3 'UNIVERSE)
C2
```

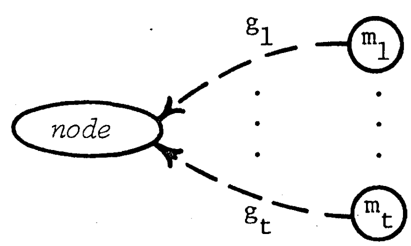



```
?(DAP 'C3 'TX 'C4 'EAST)
C3

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

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

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

(DIG *node*)[1] Destroy Inpointing edGes \'dig\

Informal Definition

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



error condition:

- *node* does not exist

Formal Definition

DIG[n] = n

with effects:

NGN := NGN − {(m g n)|m ∊ N, g ∊ G}

NGNSV := NGNSV − {(((m g n) s) v)|m ∊ N, g ∊ G, s ∊ S, v ∊ V}

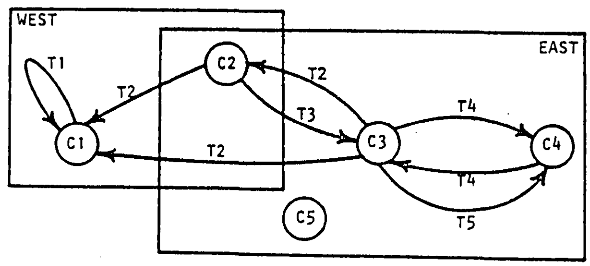
error condition:

- n ∉ N

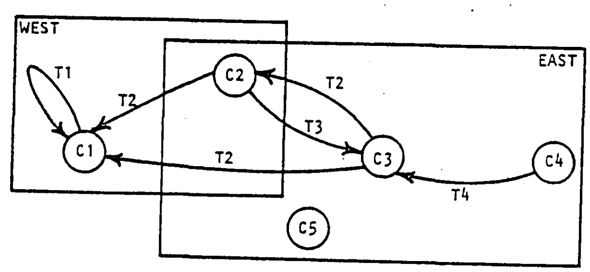
[1] See alternative form on page 92.

IG

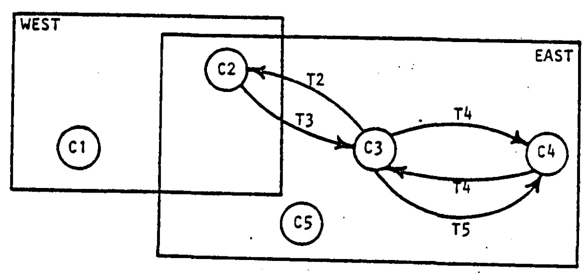
Illustrations



?(DIG 'C4)
C4



?(DIG 'C1)
C1



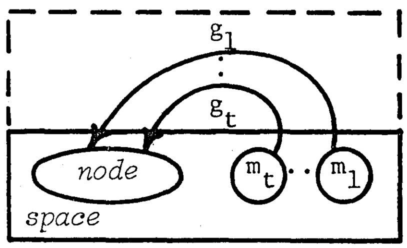
?(DIG 'C5)
C5

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

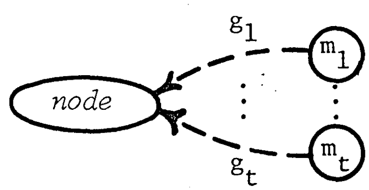
(DIG *node space*)[1] Destroy Inpointing edGes \'dig\

Informal Definition

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



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DIG[n,s] = n

with effects:

if s = UNIVERSE

then DIG[n]

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

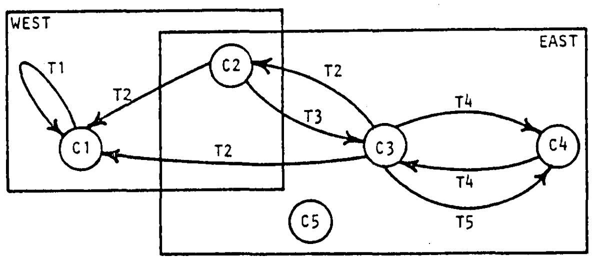
error conditions:

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

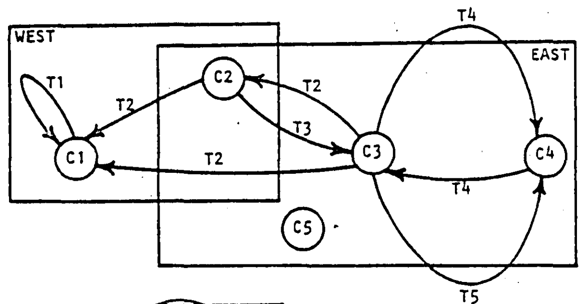
[1]See alternative form on page 90.

IG

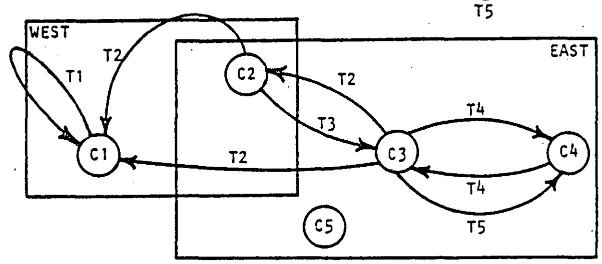
Illustrations



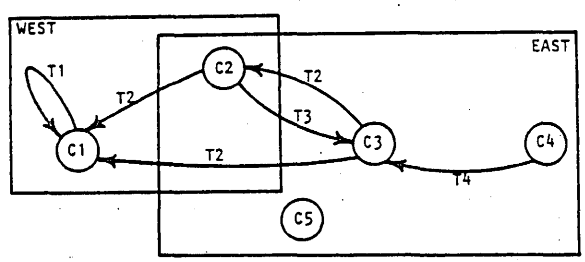
?(DIG 'C4 'EAST)
C4



?(DIG 'C1 'WEST)
C1



?(DIG 'C4 'UNIVERSE)
C4



?(DIG 'C5 'EAST)
C5

?(DIG 'CX 'EAST)

*** DIG ERROR: CX IS NOT A NODE IN SPACE EAST

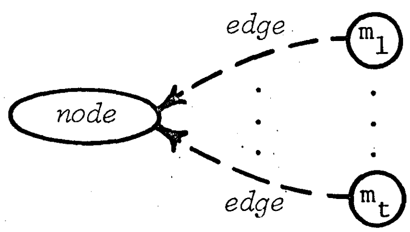
?(DIG 'C4 'SX)

*** DIG ERROR: SX IS NOT A SPACE

(DIGG *node edge*)[1] Destroy Inpointing edGes given an edGe \'dig-gə\

Informal Definition

The pseudo-function DIGG is an EXPR which has the effect of destroying all inpointing instances of *edge* to *node*. Given *node* and *edge*, DIGG destroys all instances of *edge* that point to *node*. If no such edges exist, DIGG has no effect. DIGG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DIGG[n,g] = n

with effects:

NGN := NGN - $\{(m\ g\ n) \mid m \in N\}$

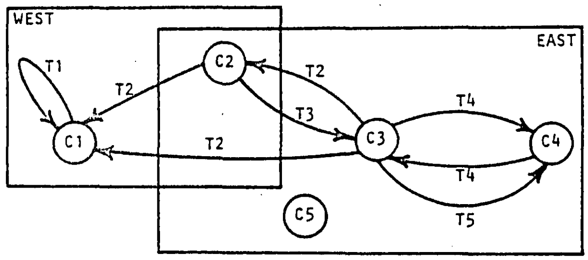
NGNSV := NGNSV - $\{(((m\ g\ n)\ s)\ v) \mid m \in N,\ s \in S,\ v \in V\}$

error condition:

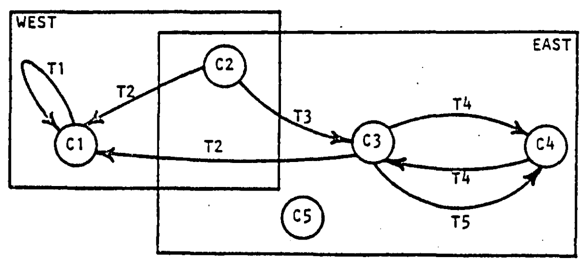
- $n \notin N$

[1]See alternative form on page 96.

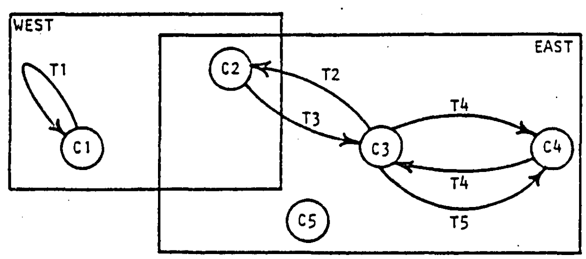
Illustrations




```
?(DIGG 'C2 'T2)
C2
```




```
?(DIGG 'C1 'T2)
C1
```

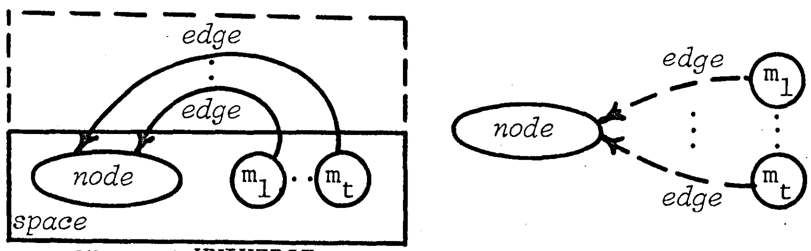



```
?(DIGG 'C5 'TX)
C5

?(DIGG 'CX 'T1)
*** DIGG ERROR: CX IS NOT A NODE
```

(DIGG *node edge space*)[1] Destroy Inpointing edGes given an edGe \'dig-gə\

Informal Definition

The pseudo-function DIGG is an EXPR which has the effect of destroying all inpointing instances of *edge* to *node* in *space*. Given *node*, *edge*, and *space*, DIGG removes all instances of *edge* from *space* that point to *node*. If *space* is UNIVERSE, DIGG removes each such *edge* from all spaces. If no such *edge* exists, DIGG has no effect. DIGG returns *node*.



space ≠ UNIVERSE *space* = UNIVERSE

error conditions:

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

Formal Definition

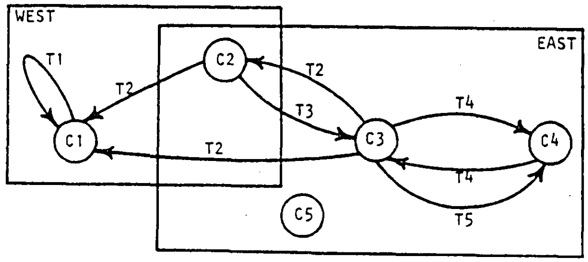
DIGG[n,g,s] = n
  with effects:
    if s = UNIVERSE
      then DIGG[n,g]
      else NGNSV := NGNSV − $\{(((m\ g\ n)\ s)\ v) \mid m \in N,\ v \in V\}$

error conditions:

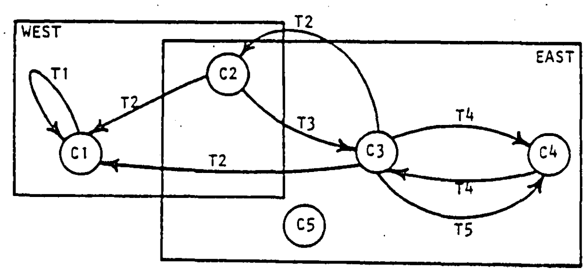
- $((n\ s)\ v) \notin NSV$ for all $v \in V$
- $s \notin S$

[1]See alternative form on page 94.

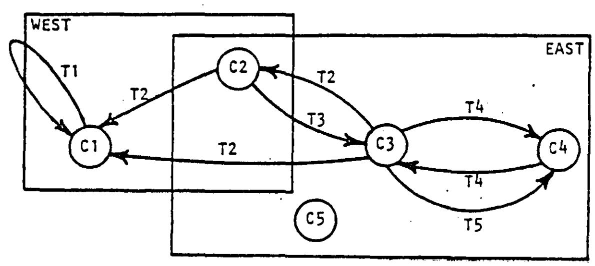
Illustrations



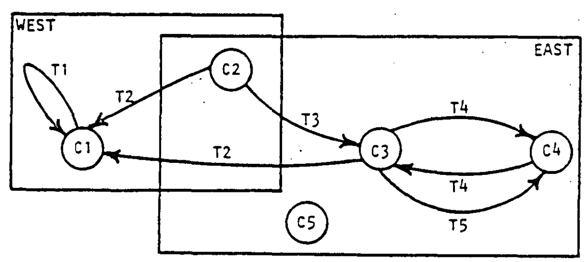
?(DIGG 'C2 'T2 'EAST)
C2



?(DIGG 'C1 'T1 'WEST)
C1



?(DIGG 'C2 'T2 'UNIVERSE)
C2



?(DIGG 'C2 'TX 'EAST)
C2

?(DIGG 'CX 'T2 'EAST)

*** DIGG ERROR: CX IS NOT A NODE IN SPACE EAST

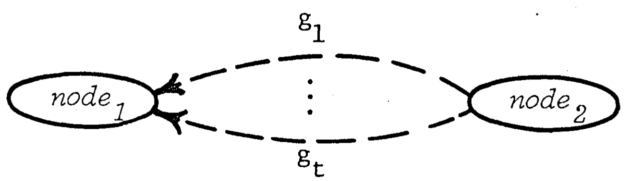
?(DIGG 'C2 'T2 'SX)

*** DIGG ERROR: SX IS NOT A SPACE

(DIGN *node*$_1$ *node*$_2$)[1] Destroy Inpointing edGes given a Node \'dig-in\

Informal Definition

The pseudo-function DIGN is an EXPR which has the effect of destroying all inpointing edges of *node*$_1$ that originate from *node*$_2$. Given *node*$_1$ and *node*$_2$, DIGN destroys all edges g_i where for each i, g_i points from *node*$_2$ to *node*$_1$. If no such edges exist, DIGN has no effect. DIGN returns *node*$_1$.



error conditions:

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

Formal Definition

DIGN[n,m] = n

with effects:

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

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

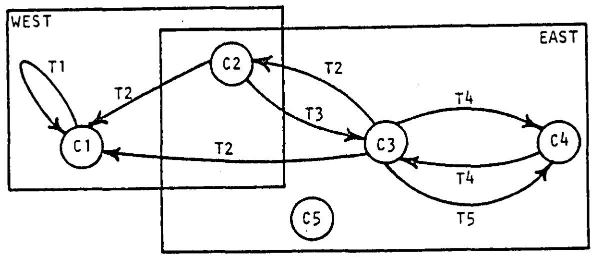
error conditions:

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

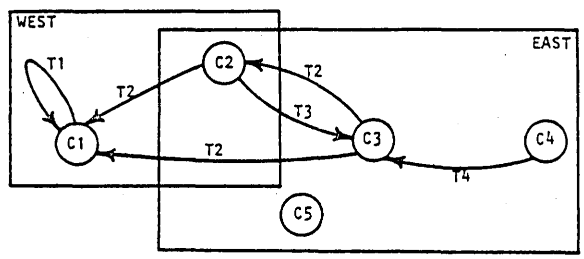
[1]See alternative form on page 100.

GN

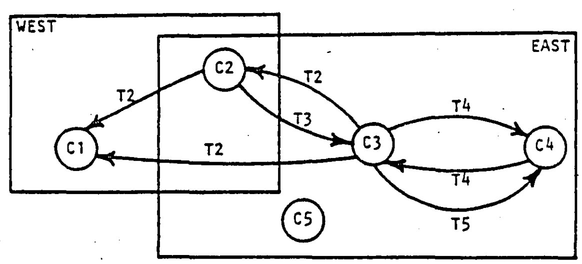
Illustrations



?(DIGN 'C4 'C3)
C4



?(DIGN 'C1 'C1)
C1



?(DIGN 'C5 'C3)
C5

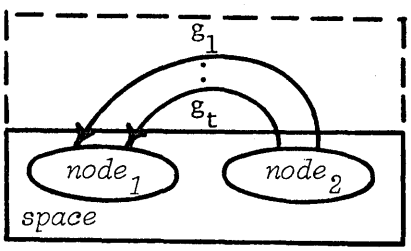
?(DIGN 'CX 'C1)
*** DIGN ERROR: CX IS NOT A NODE

?(DIGN 'C1 'CX)
*** DIGN ERROR: CX IS NOT A NODE

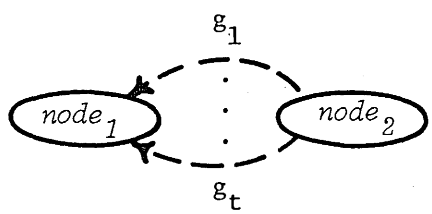
(DIGN $node_1$ $node_2$ *space*)[1] <u>D</u>estroy <u>I</u>npointing ed<u>G</u>es given a <u>N</u>ode \'dig-in\

<u>Informal Definition</u>

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

<u>Formal Definition</u>

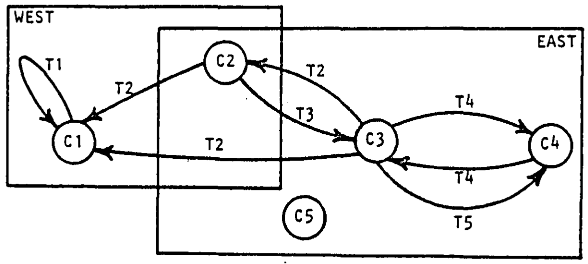
DIGN[n,m,s] = n
  with effects:
    if s = UNIVERSE
      then DIGN[n,m]
      else NGNSV := NGNSV − $\{(((m\ g\ n)\ s)\ v) \mid g \in G,\ v \in V\}$

error conditions:

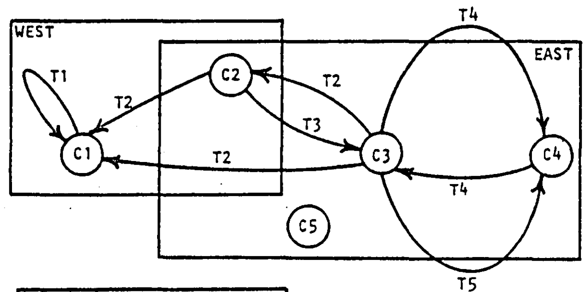
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $((m\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1] See alternative form on page 98.

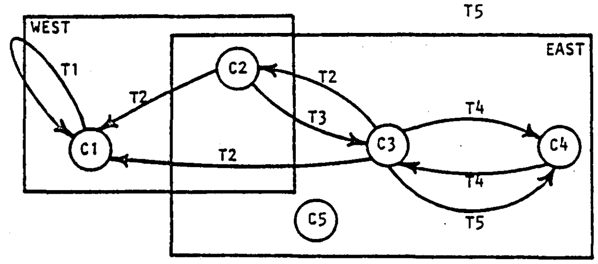
Illustrations



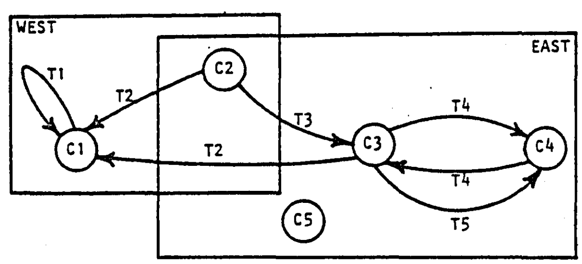

```
?(DIGN 'C4 'C3 'EAST)
C4
```




```
?(DIGN 'C1 'C1 'WEST)
C1
```




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

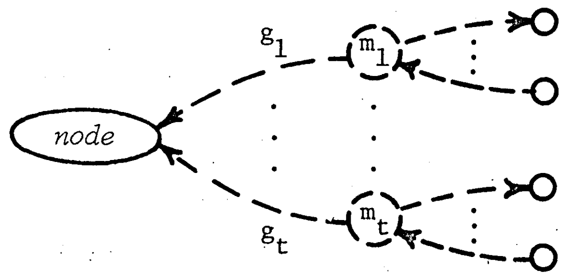



```
?(DIGN 'C5 'C4 'EAST)
C5

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

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

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

(DIN *node*)[1] Destroy Inpointing Nodes \'din\

Informal Definition

The pseudo-function DIN is an EXPR which has the effect of destroying all inpointing nodes of *node*. Given *node*, DIN destroys all nodes m_i where for each i, some edge g_i points to *node* from node m_i. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DIN has no effect. DIN returns *node*.



error condition:

- *node* does not exist

Formal Definition

DIN[n] = n
    with effects:
        for each $m \in$ SIN[n]
            DUN[m]

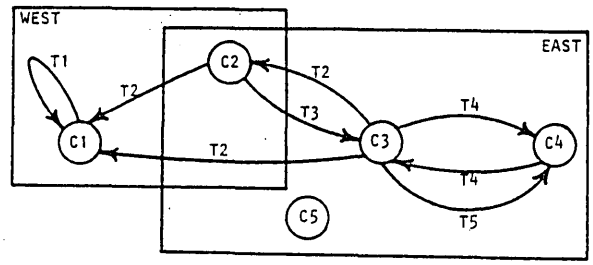
error condition:

- $n \notin N$

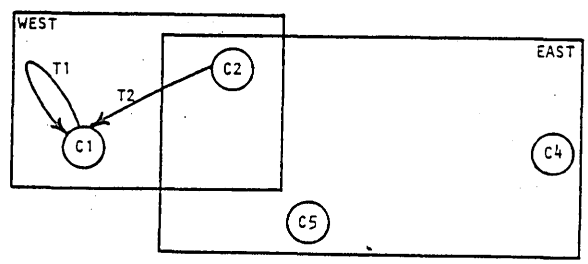
[1] See alternative form on page 104.

[2] See DAG on page 66.

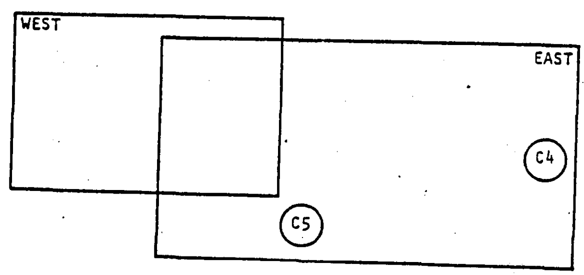
Illustrations




```
?(DIN 'C2)
C2
```




```
?(DIN 'C1)
C1
```

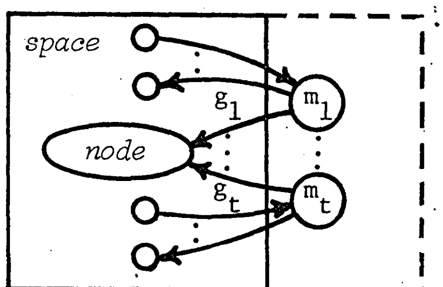



```
?(DIN 'C5)
C5

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

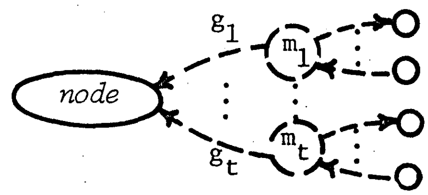
(DIN *node space*)[1] Destroy Inpointing Nodes \'din\

Informal Definition

The pseudo-function DIN is an EXPR which has the effect of destroying all inpointing nodes of *node* in *space*. Given *node* and *space*, DIN removes all nodes m_i from *space* where for each i, some edge g_i points to *node* from node m_i in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DIN removes each node m_i and all of its adjacent edges from all spaces. If no such nodes exist, DIN has no effect. DIN returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DIN[n,s] = n
  with effects:
    for each m ∊ SIN[n,s]
      DUN[m,s]

error conditions:

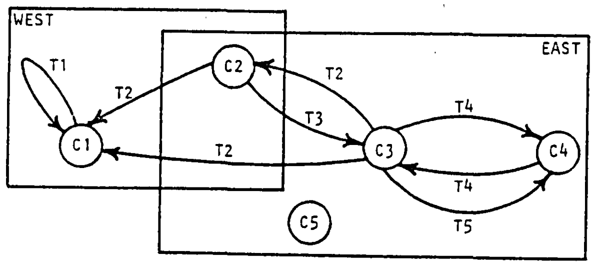
- ((n s) v) ∉ NSV for all v ∊ V
- s ∉ S

---

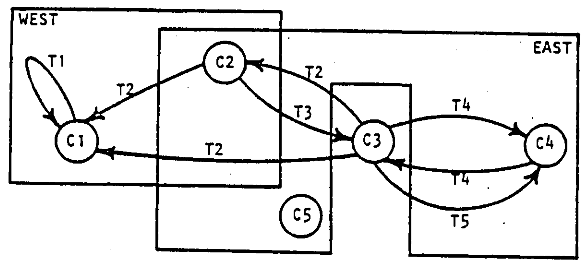
[1] See alternative form on page 102.

[2] See DAG on page 68.

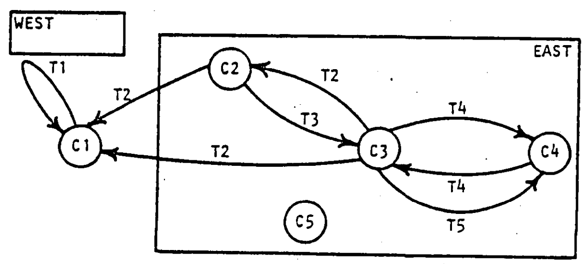
Illustrations



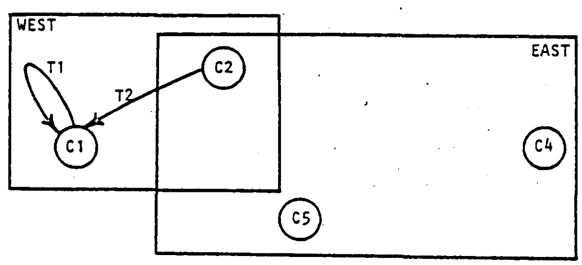
?(DIN 'C2 'EAST)
C2



?(DIN 'C1 'WEST)
C1



?(DIN 'C4 'UNIVERSE)
C4



?(DIN 'C5 'EAST)
C5

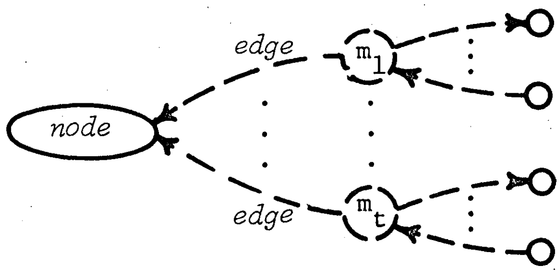
?(DIN 'CX 'EAST)
*** DIN ERROR: CX IS NOT A NODE IN SPACE EAST

?(DIN 'C2 'SX)
*** DIN ERROR: SX IS NOT A SPACE

(DING *node edge*)[1] <u>D</u>estroy <u>I</u>npointing <u>N</u>odes given an ed<u>G</u>e \'diŋ\

Informal Definition

The pseudo-function DING is an EXPR which has the effect of destroying all inpointing nodes of *node* along *edge*. Given *node* and *edge*, DING destroys all nodes m_i where for each i, *edge* points to *node* from node m_i. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DING has no effect. DING returns *node*.



error condition:

- *node* does not exist

Formal Definition

DING[n,g] = n
with effects:
for each m ∈ SING[n,g]
DUN[m]

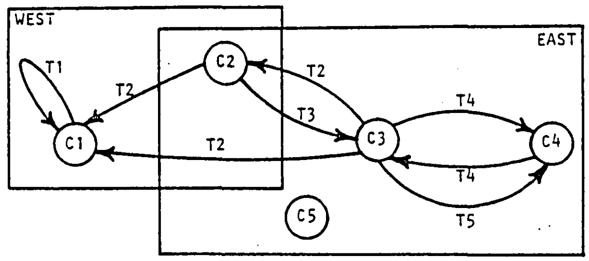
error condition:

- n ∉ N

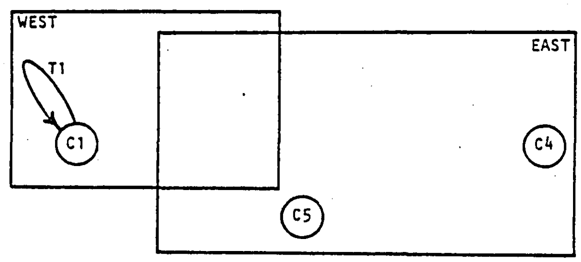
[1]See alternative form on page 108.

[2]See DAG on page 66.

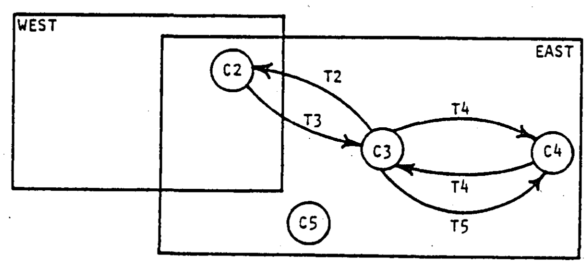
Illustrations



?(DING 'C1 'T2)
C1



?(DING 'C1 'T1)
C1



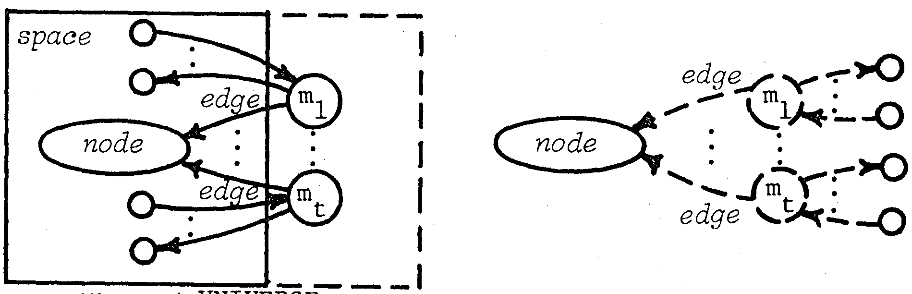
?(DING 'C5 'TX)
C5

?(DING 'CX 'T1)
*** DING ERROR: CX IS NOT A NODE

(DING *node edge space*)[1] Destroy Inpointing Nodes given an edGe \'diŋ\

Informal Definition

The pseudo-function DING is an EXPR which has the effect of destroying all inpointing nodes of *node* along *edge* in *space*. Given *node*, *edge*, and *space*, DING removes all nodes m_i from *space* where for each i, *edge* points to *node* from node m_i in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DING removes each node m_i and all its adjacent edges from all spaces. If no such nodes exist, DING has no effect. DING returns *node*.



space ≠ UNIVERSE *space* = UNIVERSE

error conditions:

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

Formal Definition

DING[n,g,s] = n
  with effects:
    for each m ∊ SING[n,g]
      DUN[m,s]

error conditions:

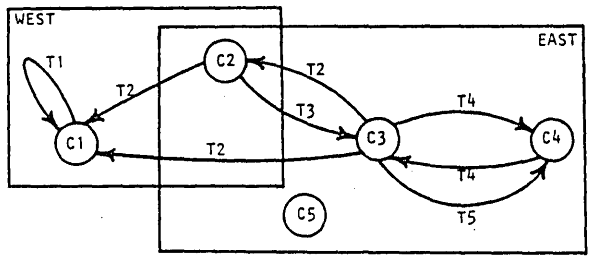
- ((n s) v) ∉ NSV for all v ∊ V
- s ∉ S

[1]See alternative form on page 106.

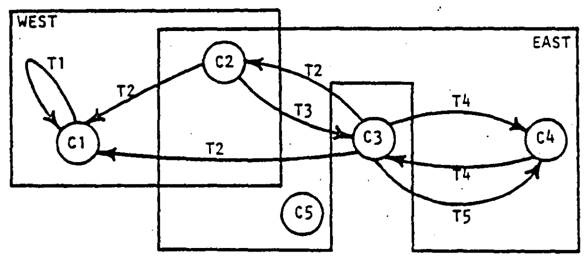
[2]See DAG on page 68.

ING

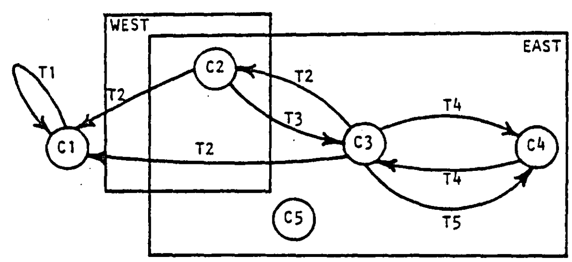
Illustrations



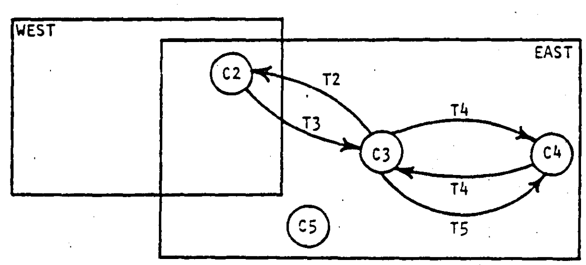

```
?(DING 'C2 'T2 'EAST)
C2
```




```
?(DING 'C1 'T1 'WEST)
C1
```




```
?(DING 'C1 'T1 'UNIVERSE)
C1
```

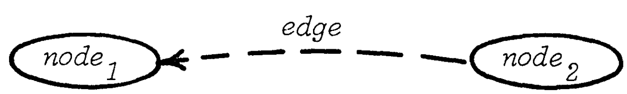



```
?(DING 'C2 'TX 'EAST)
C2

?(DING 'CX 'T2 'EAST)
*** DING ERROR: CX IS NOT A NODE IN SPACE EAST

?(DING 'C2 'T2 'SX)
*** DING ERROR: SX IS NOT A SPACE
```

(DIP $node_1$ *edge* $node_2$)[1] Destroy Inpointing Pair \'dip\

Informal Definition

The pseudo-function DIP is an EXPR which has the effect of destroying the inpointing pair (*edge* $node_2$) of $node_1$. Given $node_1$, *edge*, and $node_2$, DIP destroys *edge* from $node_2$ to $node_1$. If the inpointing pair does not exist, DIP has no effect. DIP returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

DIP[n,g,m] = n

with effects:

NGN := NGN - {(m g n)}

NGNSV := NGNSV - {(((m g n) s) v)|v $\in$ V}

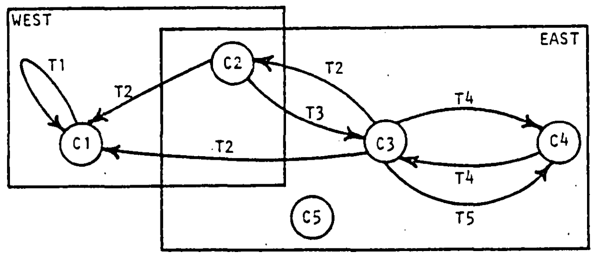
error conditions:

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

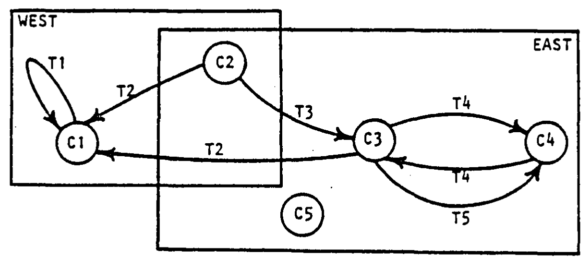
---

[1]See alternative form on page 112.

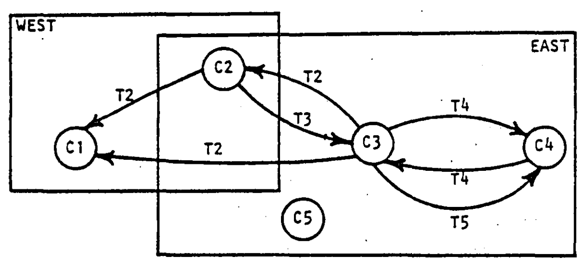
Illustrations




```
?(DIP 'C2 'T2 'C3)
C2
```




```
?(DIP 'C1 'T1 'C1)
C1
```

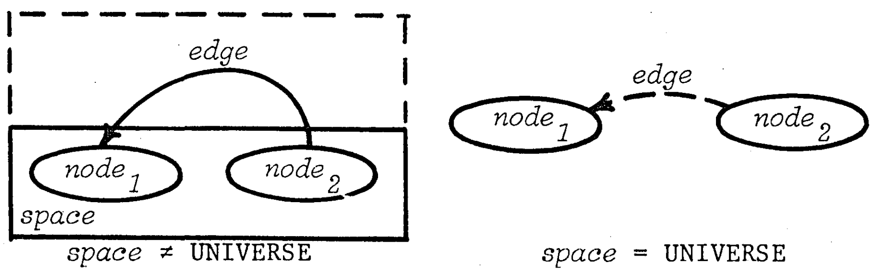



```
?(DIP 'C1 'TX 'C3)
C1

?(DIP 'CX 'T2 'C3)
*** DIP ERROR: CX IS NOT A NODE

?(DIP 'C1 'T2 'CX)
*** DIP ERROR: CX IS NOT A NODE
```

(DIP $node_1$ *edge* $node_2$ *space*)[1] Destroy Inpointing Pair \'dip\

Informal Definition

The pseudo-function DIP is an EXPR which has the effect of destroying the inpointing pair (*edge* $node_2$) of $node_1$ in *space*. Given $node_1$, *edge*, $node_2$, and *space*, DIP removes *edge* pointing from $node_2$ to $node_1$ from *space*. If *space* is UNIVERSE, DIP removes *edge* pointing from $node_1$ to $node_2$ from all spaces. If the inpointing pair does not exist in *space*, DIP has no effect. DIP returns $node_1$.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- *space* does not exist

Formal Definition

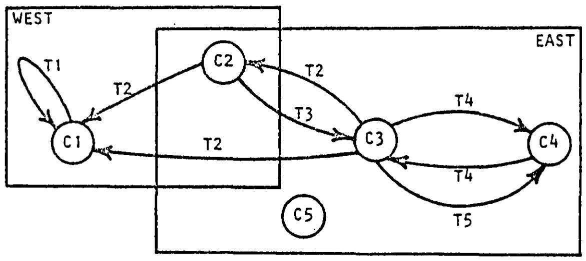
DIP[n,g,m,s] = n
  with effects:
    if s = UNIVERSE
      then DIP[n,g,m]
      else NGNSV := NGNSV − $\{(((m\ g\ n)\ s)\ v) \mid v \in V\}$

error conditions:

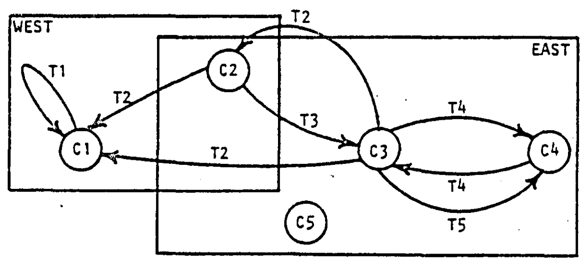
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $((m\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 100.

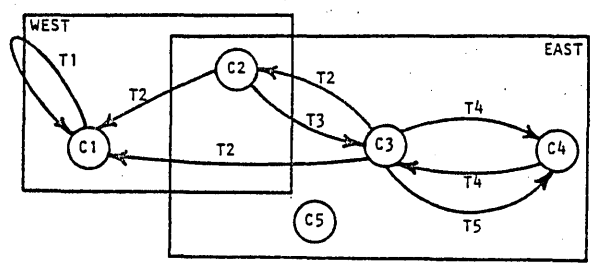
Illustrations



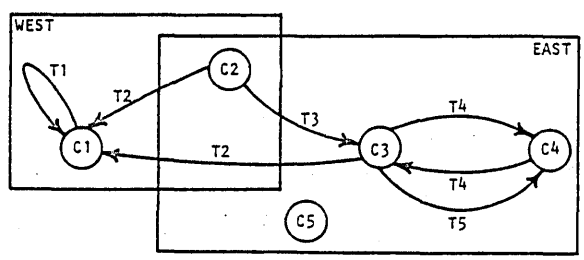

```
?(DIP 'C2 'T2 'C3 'EAST)
C2
```




```
?(DIP 'C1 'T1 'C1 'WEST)
C1
```




```
?(DIP 'C2 'T2 'C3 'UNIVERSE)
C2
```

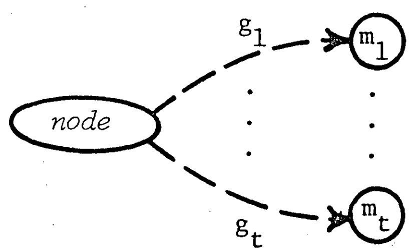



```
?(DIP 'C2 'TX 'C3 'EAST)
C2

?(DIP 'CX 'T2 'C3 'EAST)
*** DIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(DIP 'C2 'T2 'CX 'EAST)
*** DIP ERROR: CX IS NOT A NODE IN SPACE EAST

?(DIP 'C2 'T2 'C3 'SX)
*** DIP ERROR: SX IS NOT A SPACE
```

(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) | g ∈ G, m ∈ N}

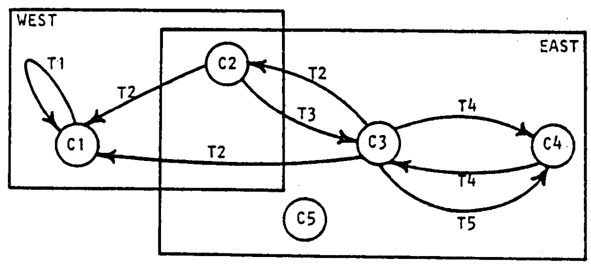
NGNSV := NGNSV − {(((n g m) s) v) | g ∈ G, m ∈ N, s ∈ S, v ∈ V}

error condition:

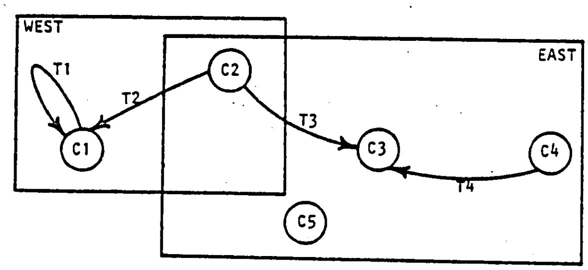
- n ∉ N

[1] See alternative form on page 116.

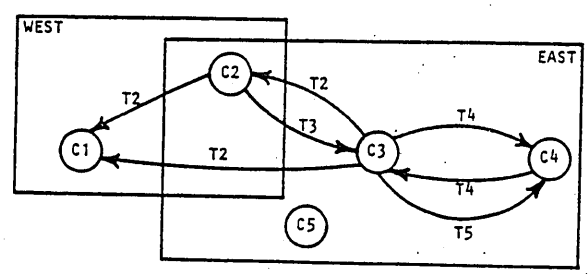
Illustrations



?(DOG 'C3)
C3



?(DOG 'C1)
C1



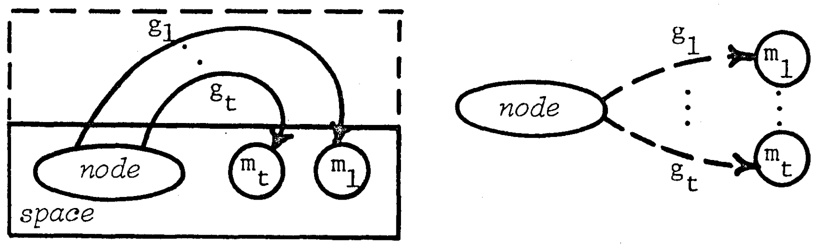
?(DOG 'C5)
C5

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

(DOG *node space*)[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* 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]

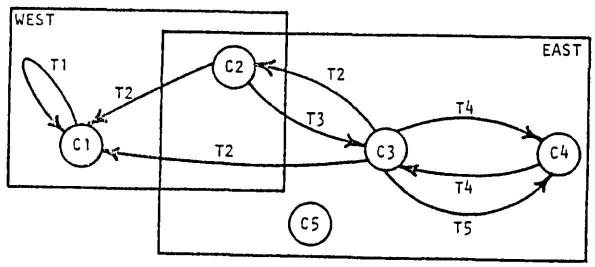
else NGNSV := NGNSV − $\{(((n\ g\ m)\ s)\ v) \mid g \in G,\ m \in N,\ v \in V\}$

error conditions:

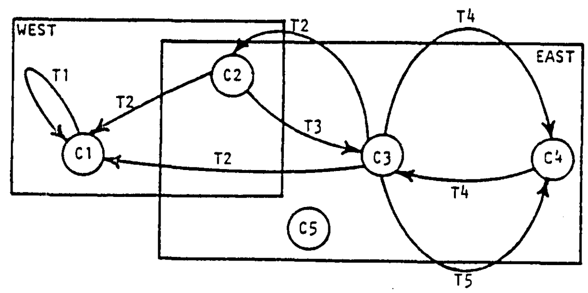
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 114.

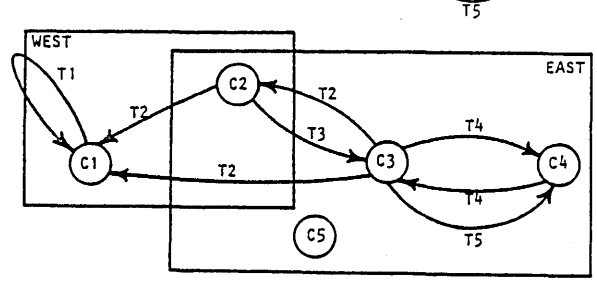
Illustrations



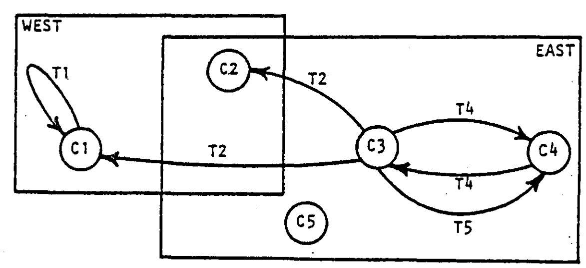

```
?(DOG 'C3 'EAST)
C3
```




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




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

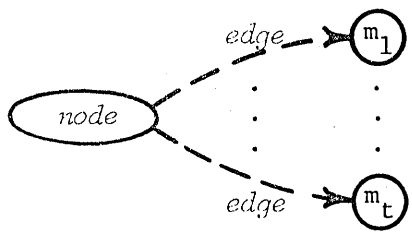



```
?(DOG 'C5 'EAST)
C5

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

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

(DOGG *node edge*)[1] Destroy Outpointing edGes given an edGe \'dog-gə\

Informal Definition

The pseudo-function DOGG is an EXPR which has the effect of destroying all outpointing instances of *edge* from *node*. Given *node* and *edge*, DOGG destroys all instances of *edge* that point away from *node*. If no such edges exist, DOGG has no effect. DOGG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DOGG[n,g] = n

with effects:

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

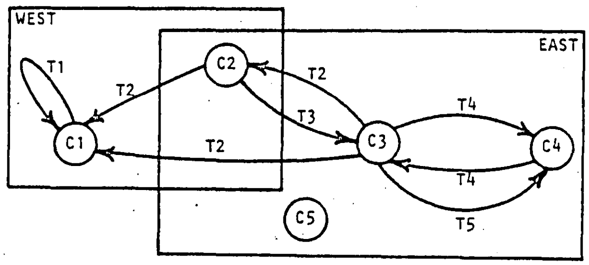
NGNSV := $\{(((n\ g\ m)\ s)\ v) \mid m \in N,\ s \in S,\ v \in V\}$

error condition:

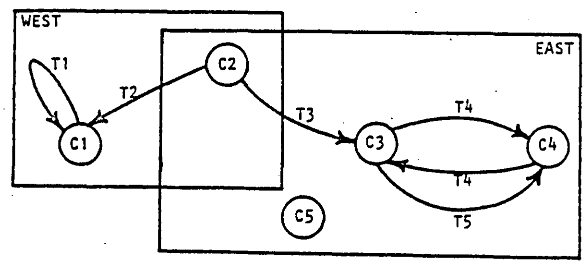
- $n \notin N$

[1] See alternative form on page 120.

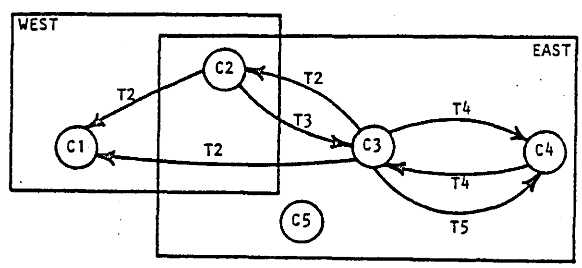
Illustrations



?(DOGG 'C3 'T2)
C3



?(DOGG 'C1 'T1)
C1



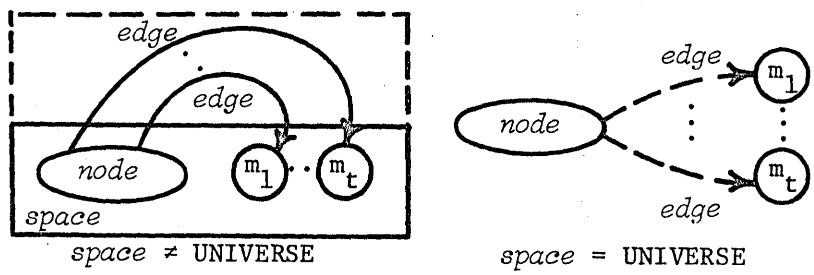
?(DOGG 'C5 'TX)
C5

?(DOGG 'CX 'T1)
*** DOGG ERROR: CX IS NOT A NODE

(DOGG *node edge space*)[1] Destroy Outpointing edGes given an edGe \'dȯg-gə\

Informal Definition

The pseudo-function DOGG is an EXPR which has the effect of destroying all outpointing instances of *edge* from *node* in *space*. Given *node*, *edge*, and *space*, DOGG removes all instances of *edge* from *space* that point away from *node*. If *space* is UNIVERSE, DOGG removes each such *edge* from all spaces. If no such *edge* exists, DOGG has no effect. DOGG returns *node*.



error conditions:

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

Formal Definition

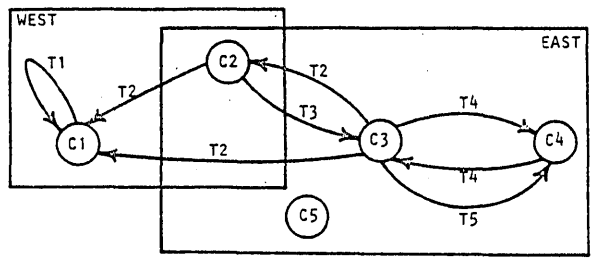
DOGG[n,g,s] = n
with effects:
if s = UNIVERSE
then DOGG[n,g]
else NGNSV := NGNSV − $\{(((n\ g\ m)\ s)\ v) \mid m \in N,\ v \in V\}$

error conditions:

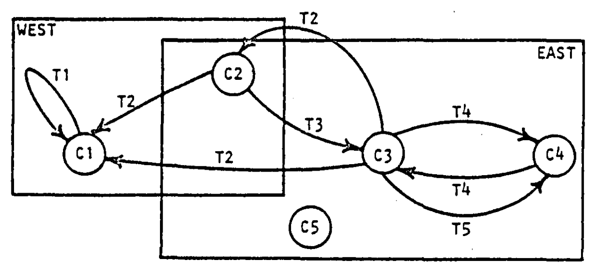
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 118.

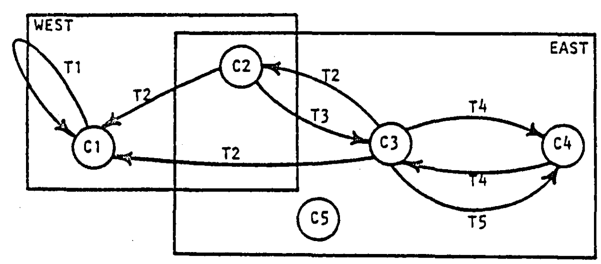
Illustrations



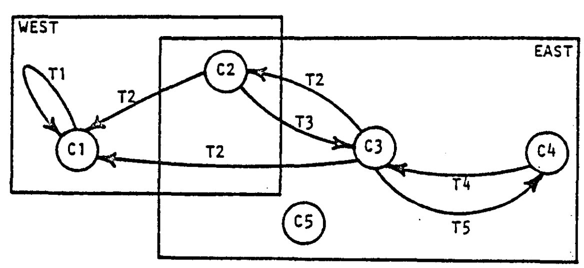

```
?(DOGG 'C3 'T2 'EAST)
C3
```




```
?(DOGG 'C1 'T1 'WEST)
C1
```




```
?(DOGG 'C3 'T4 'UNIVERSE)
C3
```

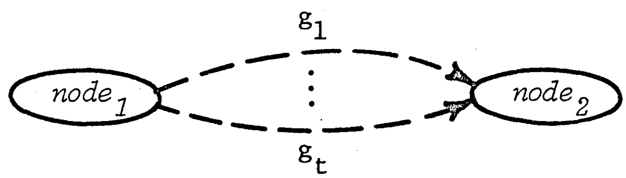



```
?(DOGG 'C3 'TX 'EAST)
C3

?(DOGG 'CX 'T2 'EAST)
*** DOGG ERROR: CX IS NOT A NODE IN SPACE EAST

?(DOGG 'C3 'T2 'SX)
*** DOGG ERROR: SX IS NOT A SPACE
```

(DOGN $node_1$ $node_2$)[1] Destroy Outpointing edGes given a Node \dǒg-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 $node_2$. Given $node_1$ and $node_2$, DOGN destroys all edges g_i where for each i, g_i points from $node_1$ to $node_2$. If no such edges exist, DOGN has no effect. DOGN returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

DOGN[n,m] = n

with effects:

NGN := NGN − {(n g m) | g ∈ G}

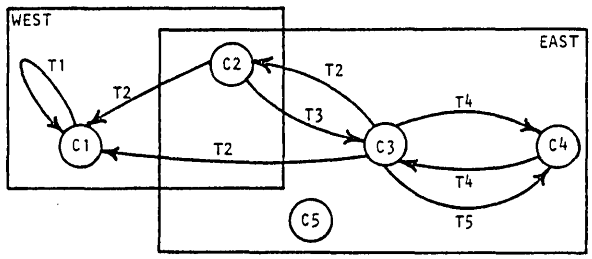
NGNSV := NGNSV − {(((n g m) s) v) | g ∈ G, s ∈ S, v ∈ V}

error conditions:

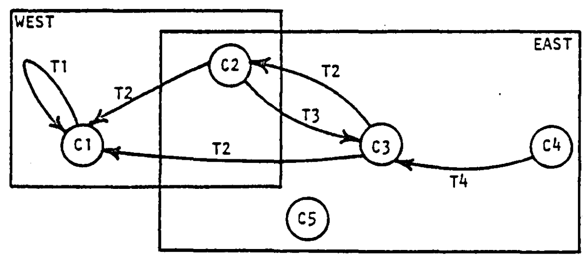
- n ∉ N
- m ∉ N

[1] See alternative form on page 124.

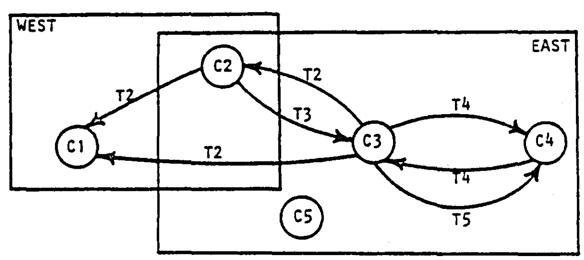
Illustrations



?(DOGN 'C3 'C4)
C3



?(DOGN 'C1 'C1)
C1



?(DOGN 'C5 'C3)
C5

?(DOGN 'CX 'C3)

*** DOGN ERROR: CX IS NOT A NODE

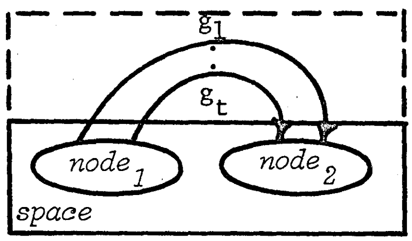
?(DOGN 'C3 'CX)

*** DOGN ERROR: CX IS NOT A NODE

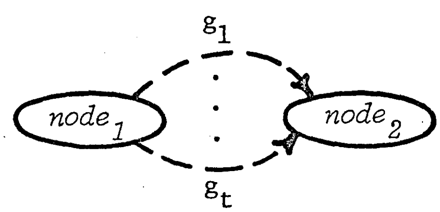
(DOGN $node_1$ $node_2$ *space*)[1] Destroy Outpointing edGes given a Node \'dȯg-in\

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]

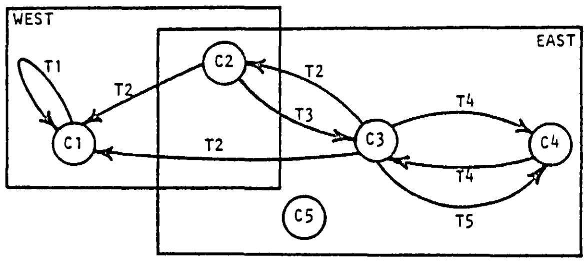
else NGNSV := NGNSV − $\{(((n\ g\ m)\ s)\ v) \mid g \in G,\ v \in V\}$

error conditions:

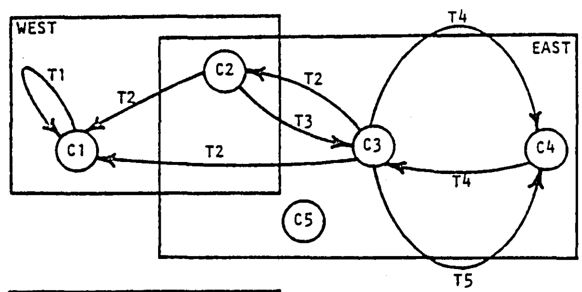
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $((m\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1] See alternative form on page 122.

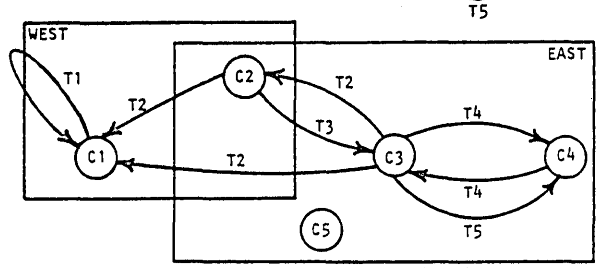
Illustrations



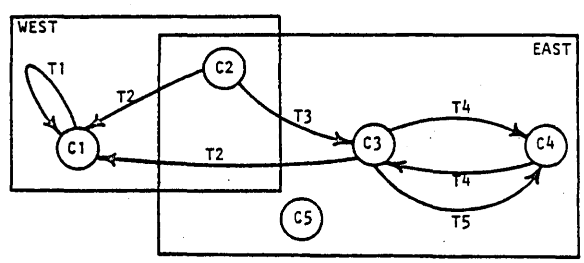

```
?(DOGN 'C3 'C4 'EAST)
C3
```




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




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

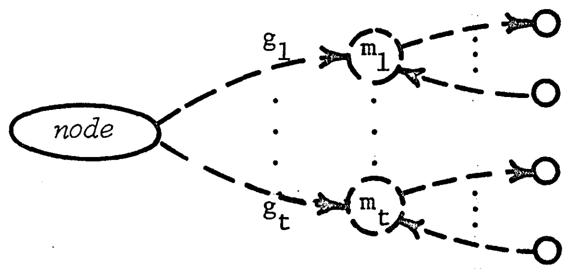



```
?(DOGN 'C5 'C4 'EAST)
C5

?(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
```

(DON *node*)[1] Destroy Outpointing Nodes \'dän\

Informal Definition

The pseudo-function DON is an EXPR which has the effect of destroying all outpointing nodes of *node*. Given *node*, DON destroys all nodes m_i where for each i, some edge g_i points to node m_i from *node*. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DON has no effect. DON returns *node*.



error condition:

- *node* does not exist

Formal Definition

DON[n] = n

with effects:

for each m $\in$ SON[n]

DUN[m]

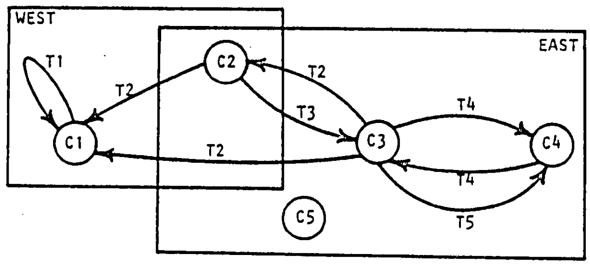
error condition:

- n $\notin$ N

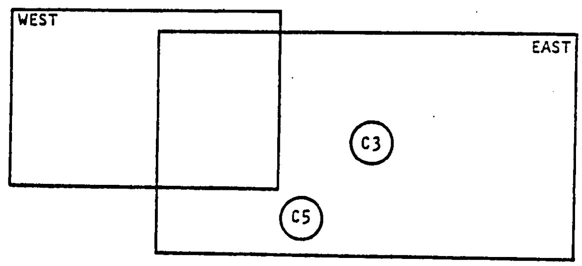
[1]See alternative form on page 128.

[2]See DAG on page 66.

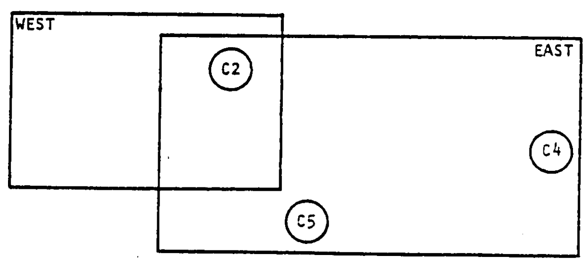
Illustrations




```
?(DON 'C3)
C3
```




```
?(DON 'C2)
C2
```

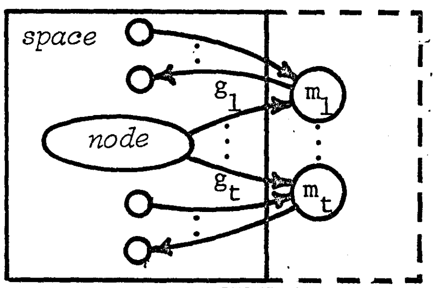



```
?(DON 'C5)
C5

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

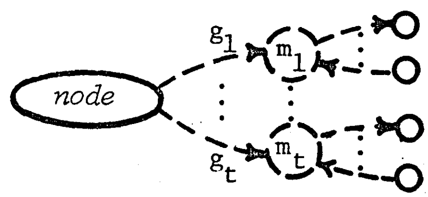
(DON *node space*)[1] Destroy Outpointing Nodes \'dän\

Informal Definition

The pseudo-function DON is an EXPR which has the effect of destroying all outpointing nodes of *node* in *space*. Given *node* and *space*, DON removes all nodes m_i from *space* where for each i, some edge g_i points to node m_i from *node* in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DON removes each node m_i and all of its adjacent edges from all spaces. If no such nodes exist, DON has no effect. DON returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DON[n,s] = n
  with effects:
    for each m ∈ SON[n,s]
      DUN[m,s]

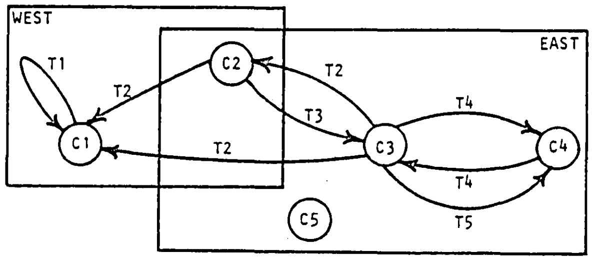
error conditions:

- $n \notin N$
- $s \notin S$

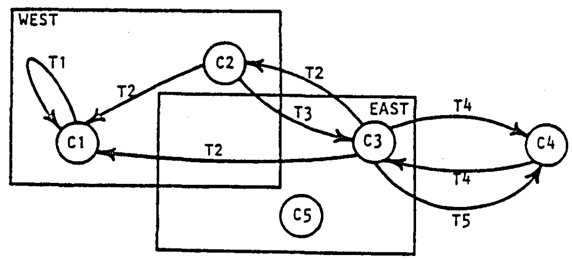
[1]See alternative form on page 126.

[2]See DAG on page 68.

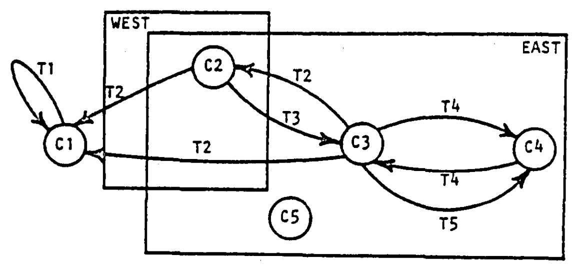
Illustrations



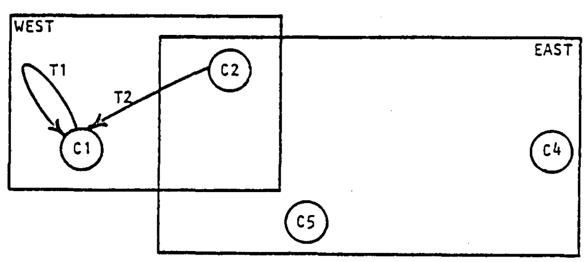

```
?(DON 'C3 'EAST)
C3
```




```
?(DON 'C2 'WEST)
C2
```




```
?(DON 'C4 'UNIVERSE)
C4
```

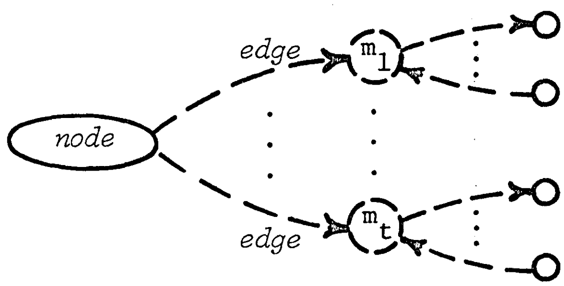



```
?(DON 'C5 'EAST)
C5

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

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

(DONG *node edge*)[1] Destroy Outpointing Nodes given an edGe \'dȯŋ\

Informal Definition

The pseudo-function DONG is an EXPR which has the effect of destroying all outpointing nodes of *node* pointed to by *edge*. Given *node* and *edge*, DONG destroys all nodes m_i where for each i, *edge* points to node m_i from *node*. Destroying each node m_i includes destroying all its adjacent edges.[2] If no such nodes exist, DONG has no effect. DONG returns *node*.



error condition:

- *node* does not exist

Formal Definition

DONG[n,g] = n

with effects:

for each m $\in$ SONG[n,g]

DUN[m]

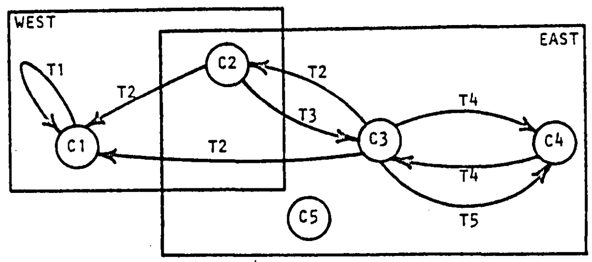
error condition:

- n $\notin$ N

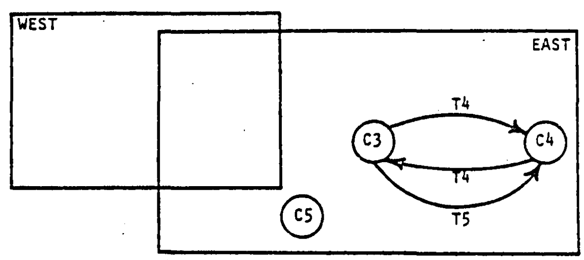
[1]See alternative form on page 132.

[2]See DAG on page 66.

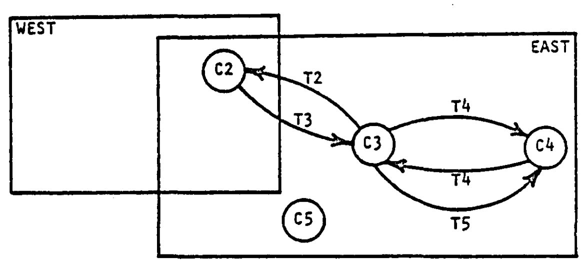
Illustrations




```
?(DONG 'C3 'T2)
C3
```




```
?(DONG 'C1 'T1)
C1
```

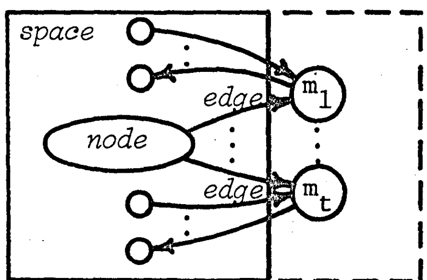



```
?(DONG 'C5 'TX)
C5

?(DONG 'CX 'T1)
*** DONG ERROR: CX IS NOT A NODE
```

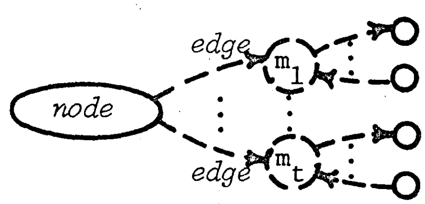
(DONG *node edge space*)[1] Destroy Outpointing Nodes given an edGe \'doŋ\

Informal Definition

The pseudo-function DONG is an EXPR which has the effect of destroying all outpointing nodes of *node* pointed to by *edge* in *space*. Given *node*, *edge*, and *space*, DONG removes all nodes m_i from *space* where for each i, *edge* points to node m_i from *node* in *space*. Removing each node m_i includes removing all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DONG removes each node m_i and all its adjacent edges from all spaces. If no such nodes exist, DONG has no effect. DONG returns *node*.



space ≠ UNIVERSE



space = UNIVERSE

error conditions:

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

Formal Definition

DONG[n,g,s] = n

with effects:

for each m ∊ SONG[n,g,s]

DUN[m,s]

error conditions:

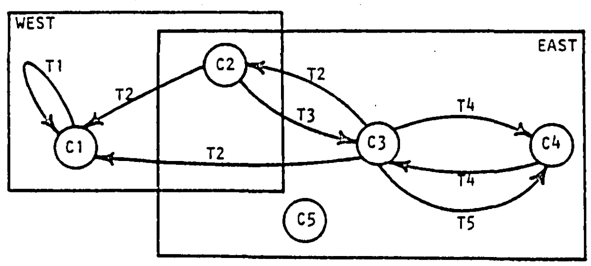
- $n \notin N$
- $s \notin S$

---

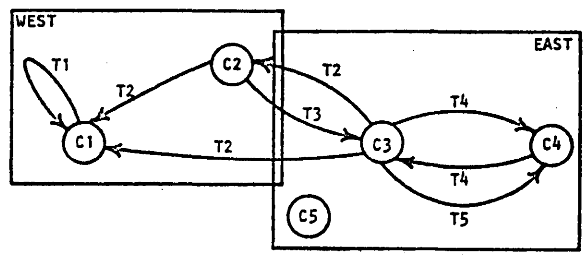
[1]See alternative form on page 130.

[2]See DAG on page 68.

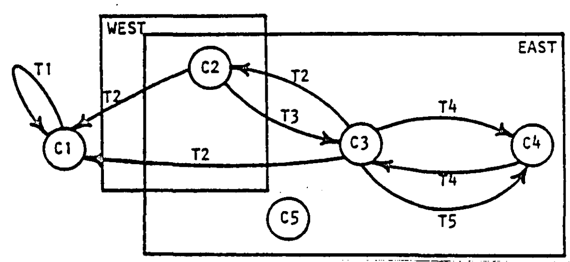
Illustrations



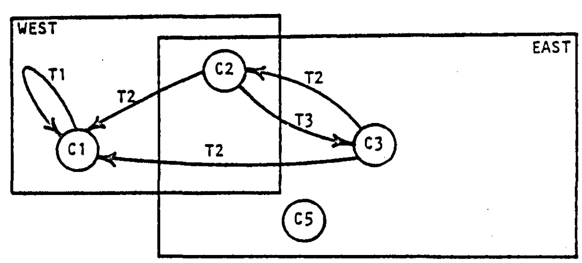
?(DONG 'C3 'T2 'EAST)
C3



?(DONG 'C1 'T1 'WEST)
C1



?(DONG 'C3 'T5 'UNIVERSE)
C3



?(DONG 'C3 'TX 'EAST)
C3

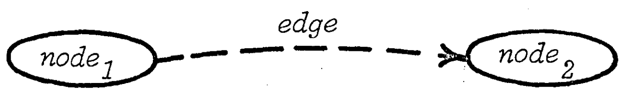
?(DONG 'CX 'T2 'EAST)
*** DONG ERROR: CX IS NOT A NODE IN SPACE EAST

?(DONG 'C3 'T2 'SX)
*** DONG ERROR: SX IS NOT A SPACE

(DOP $node_1$ *edge* $node_2$)[1] Destroy Outpointing Pair \'däp\

Informal Definition

The pseudo-function DOP is an EXPR which has the effect of destroying the outpointing pair (*edge* $node_2$) of $node_1$. Given $node_1$, *edge*, and $node_2$, DOP destroys *edge* from $node_1$ to $node_2$. If the outpointing pair does not exist, DOP has no effect. DOP returns $node_1$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

DOP[n,g,m] = n

with effects:

NGN := NGN - {(n g m)}

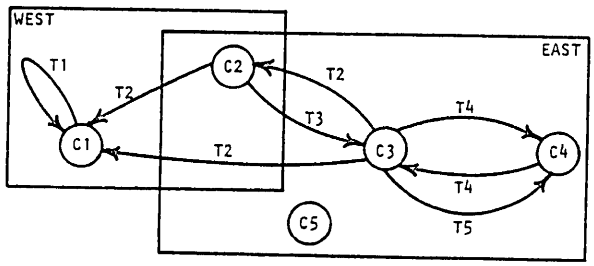
NGNSV := NGNSV - {(((n g m) s) v) | s ∈ S, v ∈ V}

error conditions:

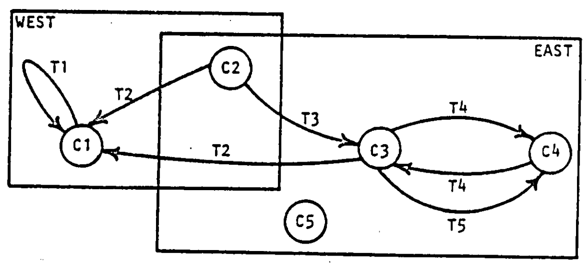
- n ∉ N
- m ∉ N

[1]See alternative form on page 136.

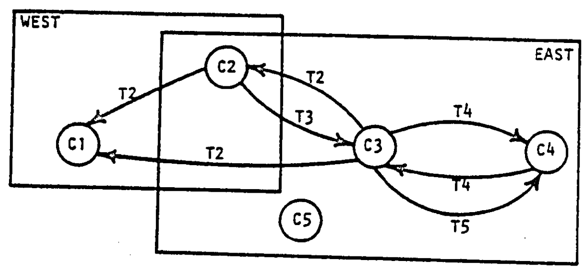
Illustrations




```
?(DOP 'C3 'T2 'C2)
C3
```




```
?(DOP 'C1 'T1 'C1)
C1
```

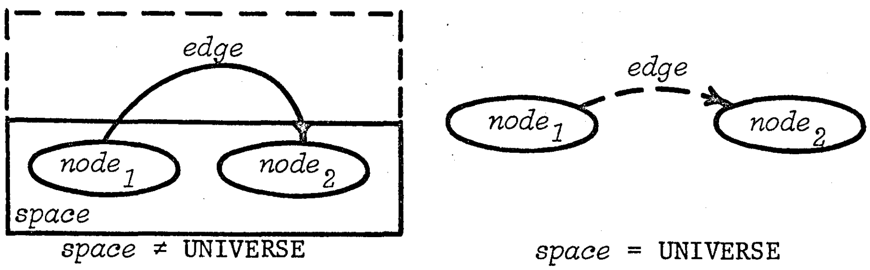



```
?(DOP 'C5 'TX 'C3)
C5

?(DOP 'CX 'T2 'C3)
*** DOP ERROR: CX IS NOT A NODE

?(DOP 'C3 'T2 'CX)
*** DOP ERROR: CX IS NOT A NODE
```

(DOP $node_1$ *edge* $node_2$ *space*)[1] Destroy Outpointing Pair \'däp\

Informal Definition

The pseudo-function DOP is an EXPR which has the effect of destroying the outpointing pair (*edge* $node_2$) of $node_1$ in *space*. Given $node_1$, *edge*, $node_2$, and *space*, DOP removes *edge* pointing from $node_1$ to $node_2$ from *space*. If *space* is UNIVERSE, DOP removes *edge* pointing from $node_1$ to $node_2$ from all spaces. If the outpointing pair does not exist in *space*, DOP has no effect. DOP returns $node_1$.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- *space* does not exist

Formal Definition

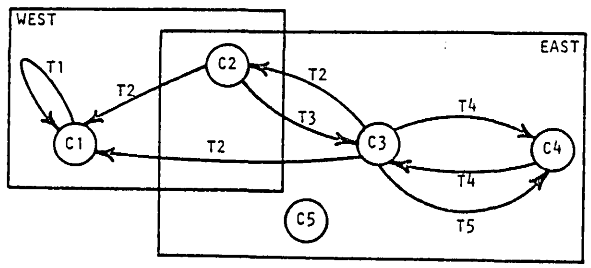
DOP[n,g,m,s] = n
with effects:
if s = UNIVERSE
then DOP[n,g,m]
else NGNSV := NGNSV - $\{(((n\ g\ m)\ s)\ v) \mid v \in V\}$

error conditions:

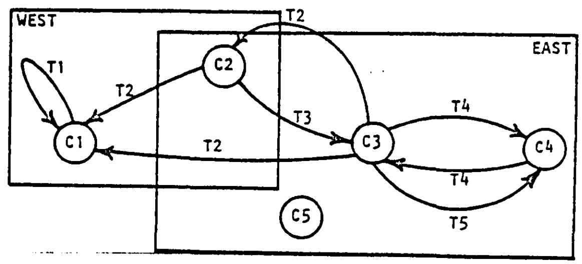
- $((n\ s)\ v) \notin$ NSV for all $v \in V$
- $((m\ s)\ v) \notin$ NSV for all $v \in V$
- $s \notin S$

[1] See alternative form on page 134.

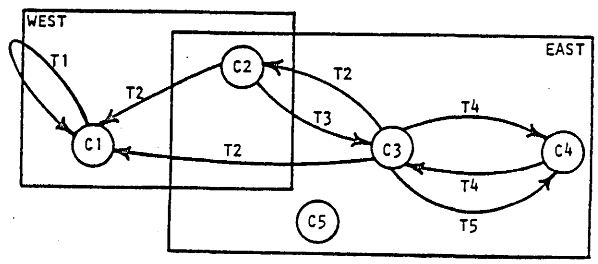
Illustrations



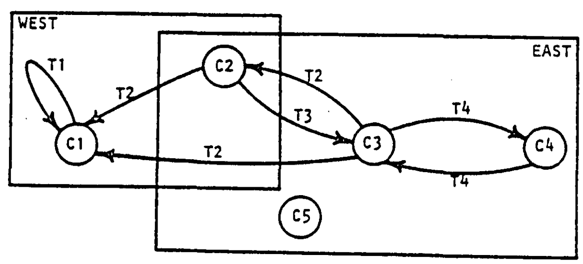

```
?(DOP 'C3 'T2 'C2 'EAST)
C3
```




```
?(DOP 'C1 'T1 'C1 'WEST)
C1
```




```
?(DOP 'C3 'T5 'C4 'UNIVERSE)
C3
```

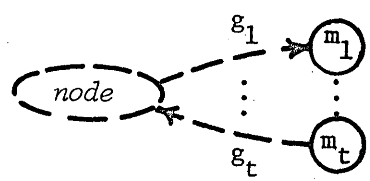



```
?(DOP 'C3 'TX 'C2 'EAST)
C3

?(DOP 'CX 'T2 'C2 'EAST)
*** DOP ERROR: CX IS NOT A NODE IN SPACE EAST

?(DOP 'C3 'T2 'CX 'EAST)
*** DOP ERROR: CX IS NOT A NODE IN SPACE EAST

?(DOP 'C3 'T2 'C2 'SX)
*** DOP ERROR: SX IS NOT A SPACE
```

(DUN *node*)[1] Destroy Unqualified Node \'dən\

Informal Definition

The pseudo-function DUN is an EXPR which has the effect of destroying *node*. Given *node*, DUN destroys *node* and all its adjacent edges.[2] DUN has no effect if *node* does not exist. DUN returns *node*.



Formal Definition

DUN[n] = n

with effects:

if n ∈ N

then DAG[n]

N := N - {n}

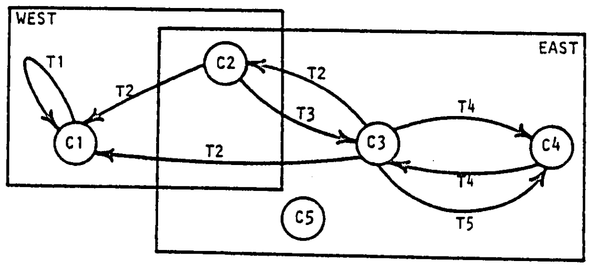
NSV := NSV - {((n s) v)|s ∈ S, v ∈ V}

---

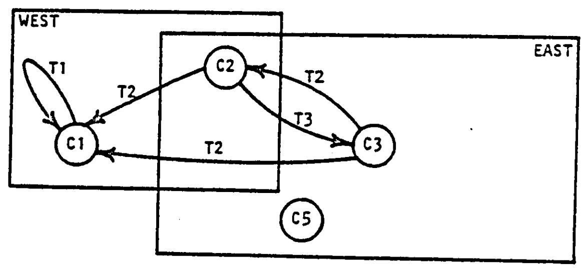
[1] See alternative form on page 140.

[2] See DAG on page 66.

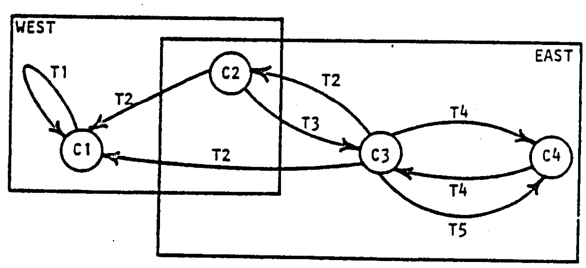
Illustrations




```
?(DUN 'C4)
C4
```




```
?(DUN 'C5)
C5
```

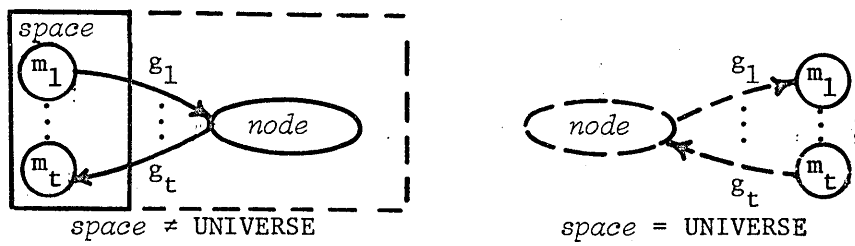



```
?(DUN 'CX)
CX
```

(DUN *node space*)[1] Destroy Unqualified Node \'dən\

Informal Definition

The pseudo-function DUN is an EXPR which has the effect of destroying *node* in *space*. Given *node* and *space*, DUN removes *node* and all its adjacent edges[2] from *space*. If *space* is UNIVERSE, DUN removes *node* and all its adjacent edges from all spaces. DUN has no effect if *node* does not exist in *space*. DUN returns *node*.



error condition:

- *space* does not exist

Formal Definition

DUN[n,s] = n

with effects:

if s = UNIVERSE then DUN[n]

else NSV := NSV − $\{((n\ s)\ v) | v \in V\}$

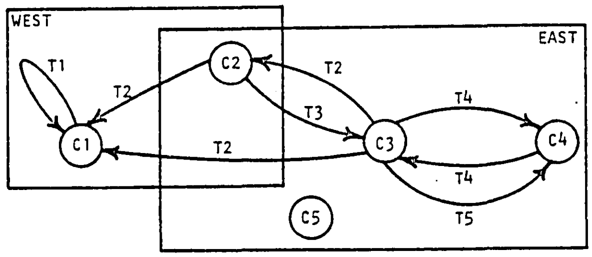
error condition:

- $s \notin S$

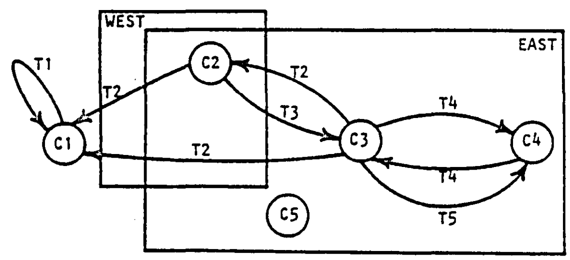
[1]See alternative form on page 138.

[2]See DAG on page 68.

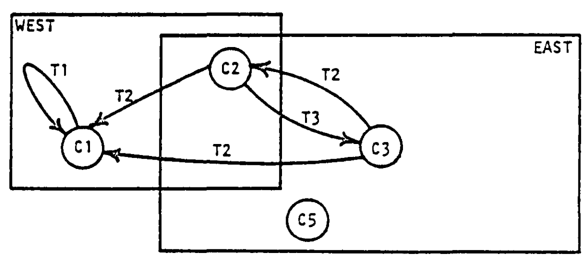
Illustrations




```
?(DUN 'C1 'WEST)
C1
```




```
?(DUN 'C4 'UNIVERSE)
C4
```

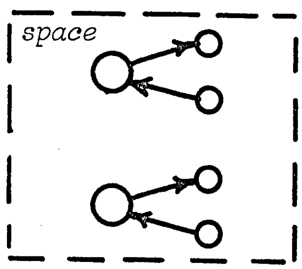



```
?(DUN 'C5 'WEST)
C5

?(DUN 'CX 'EAST)
CX

?(DUN 'C1 'SX)
*** DUN ERROR: SX IS NOT A SPACE
```

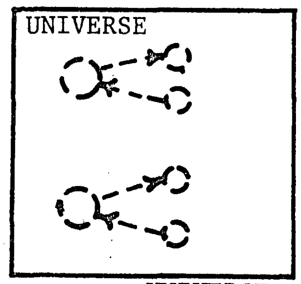
(DUS *space*) Destroy Unqualified Space \'dəs\

Informal Definition

The pseudo-function DUS is an EXPR which has the effect of destroying *space*. Given *space*, DUS removes all nodes and edges from *space*. If *space* is UNIVERSE, DUS removes all nodes and edges from all spaces and sets the value of UNIVERSE to NIL. DUS has no effect if *space* is empty except if space is UNIVERSE and has a value other than NIL. DUS returns *space*.



space ≠ UNIVERSE



space = UNIVERSE

Formal Definition

DUS[s] = s

with effects:

for each n ∈ SUN[s]

DUN[n,s]

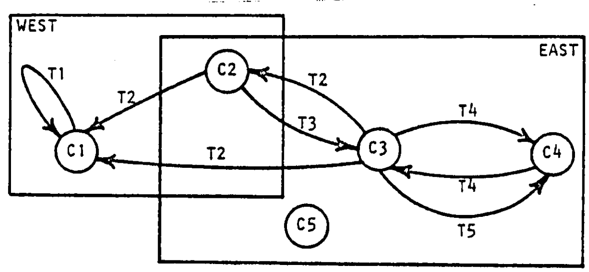
if s = UNIVERSE then BUS[UNIVERSE,NIL]

else S := S - {s}

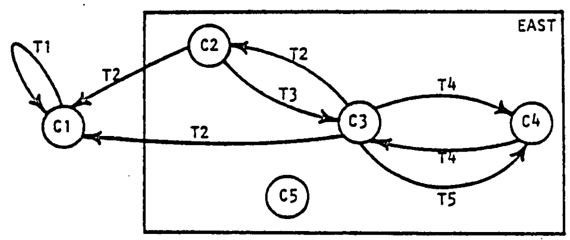
SV := SV - {(s v)|v ∈ V}

DUS

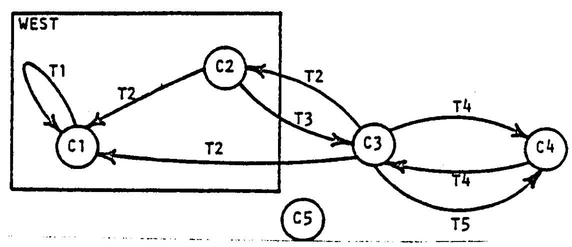
Illustrations



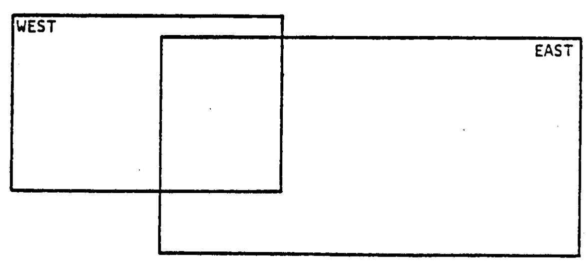
?(DUS 'WEST)
WEST



?(DUS 'EAST)
EAST



?(DUS 'UNIVERSE)
UNIVERSE

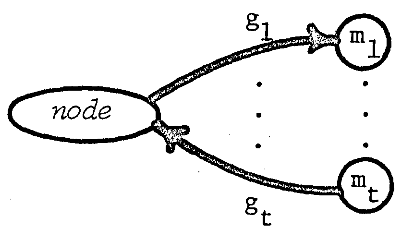


?(DUS 'SX)
SX

(SAG *node*)[1] Set of Adjacent edGes \'sag\

Informal Definition

The function SAG is an EXPR which returns the set of adjacent edges of *node*. Given *node*, SAG returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to some node m_i from *node* or points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

SAG[n] = SOG[n] ∪ SIG[n]

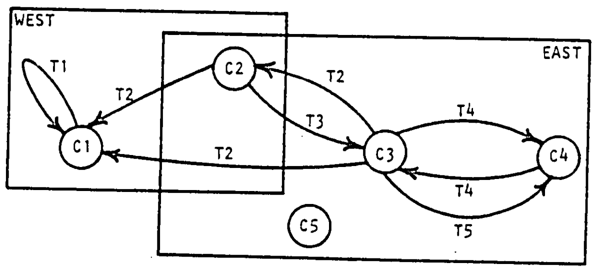
error condition:

- $n \notin N$

[1]See alternative form on page 146.

SAG

Illustrations

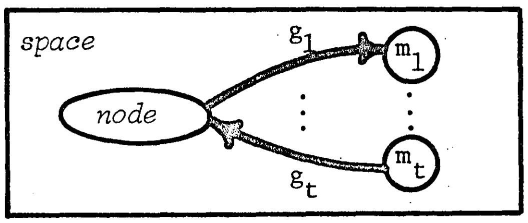



```
?(SAG 'C1)
(T1 T2)

?(SAG 'C2)
(T2 T3)

?(SAG 'C3)
(T2 T3 T4 T5)

?(SAG 'C5)
NIL

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

(SAG *node space*)[1] Set of Adjacent edGes \'sag\

Informal Definition

The function SAG is an EXPR which returns the set of adjacent edges of *node* in *space*. Given *node* and *space*, SAG returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to some node m_i from *node* in *space* or points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

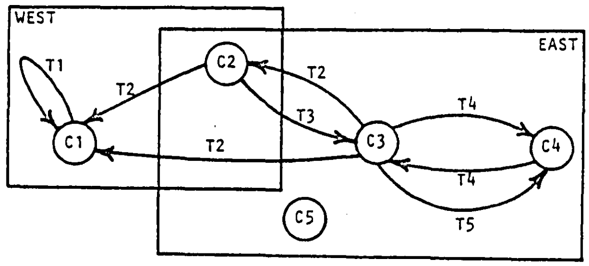
SAG[n,s] = SOG[n,s] ∪ SIG[n,s]

error conditions:

- ((n s) v) ∉ NSV for all v ∈ V
- s ∉ S

[1]See alternative form on page 144.

Illustrations

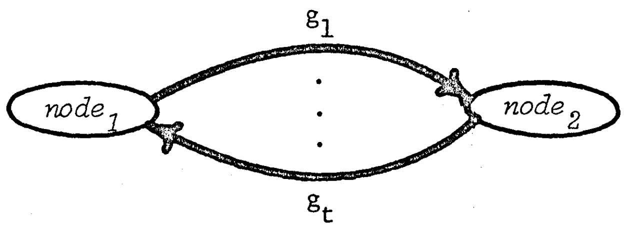



```
?(SAG 'C3 'EAST)
(T2 T3 T4 T5)

?(SAG 'C2 'WEST)
(T2)

?(SAG 'C2 'UNIVERSE)
(T2 T3)

?(SAG 'C5 'EAST)
NIL

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

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

(SAGN $node_1$ $node_2$)[1] Set of Adjacent edGes given a Node \'sag-in\

Informal Definition

The function SAGN is an EXPR which returns the set of adjacent edges of $node_1$ connected to $node_2$. Given $node_1$ and $node_2$, SAGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to $node_2$ from $node_1$ or points to $node_1$ from $node_2$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

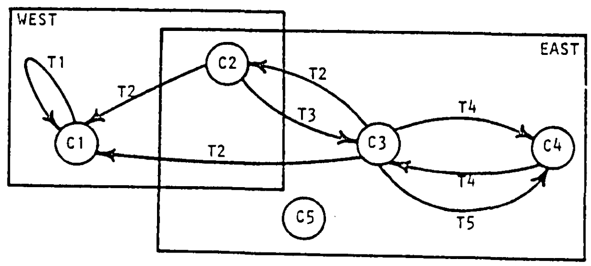
SAGN[n,m] = SOGN[n,m] ∪ SIGN[n,m]

error conditions:

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

[1]See alternative form on page 150.

Illustrations

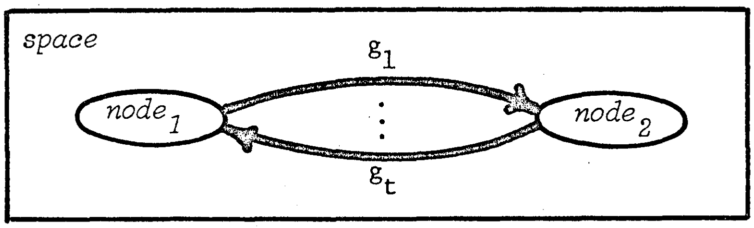



```
?(SAGN 'C1 'C2)
(T2)

?(SAGN 'C1 'C3)
(T2)

?(SAGN 'C3 'C4)
(T4 T5)

?(SAGN 'C5 'C1)
NIL

?(SAGN 'CX 'C1)
*** SAGN ERROR: CX IS NOT A NODE

?(SAGN 'C1 'CX)
*** SAGN ERROR: CX IS NOT A NODE
```

(SAGN *node*$_1$ *node*$_2$ *space*)[1] Set of Adjacent edGes given a Node \'sag-in\

Informal Definition

The function SAGN is an EXPR which returns the set of adjacent edges of *node*$_1$ connected to *node*$_2$ in *space*. Given *node*$_1$, *node*$_2$, and *space*, SAGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to *node*$_2$ from *node*$_1$ in *space* or points to *node*$_1$ from *node*$_2$ in *space*.



error conditions:

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

Formal Definition

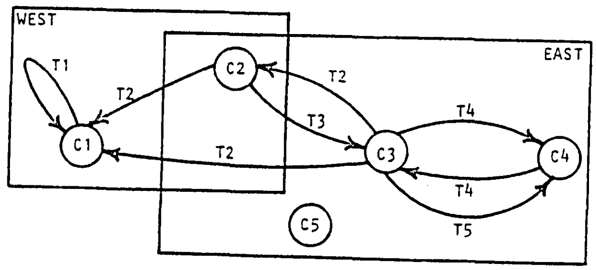
SAGN[n,m,s] = SOGN[n,m,s] $\cup$ SIGN[n,m,s]

error conditions:

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

[1]See alternative form on page 148.

Illustrations

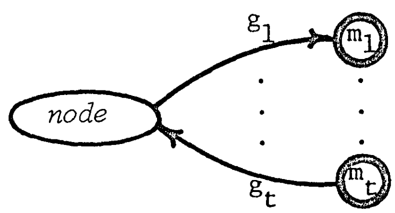



```
?(SAGN 'C2 'C3 'EAST)
(T2 T3)

?(SAGN 'C1 'C1 'WEST)
(T1)

?(SAGN 'C1 'C3 'UNIVERSE)
(T2)

?(SAGN 'C5 'C3 'EAST)
NIL

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

?(SAGN 'C2 'CX 'EAST)
*** SAGN ERROR: CX IS NOT A NODE IN SPACE EAST

?(SAGN 'C2 'C3 'SX)
*** SAGN ERROR: SX IS NOT A SPACE
```

(SAN *node*)[1] Set of Adjacent Nodes \'san\

Informal Definition

The function SAN is an EXPR which returns the set of adjacent nodes to *node*. Given *node*, SAN returns $(m_1 \ldots m_i \ldots m_t)$ where for each i, some edge g_i points to node m_i from *node* or points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

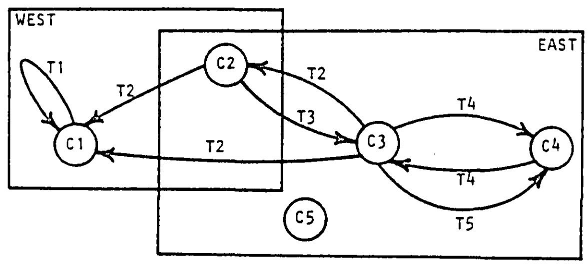
$SAN[n] = SON[n] \cup SIN[n]$

error condition:

- $n \notin N$

[1]See alternative form on page 154.

Illustrations

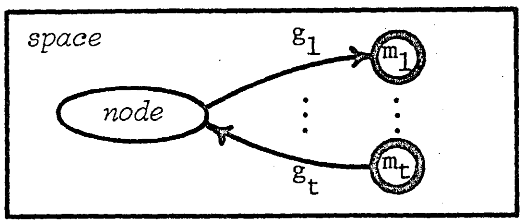



```
?(SAN 'C1)
(C1 C2 C3)

?(SAN 'C2)
(C1 C3)

?(SAN 'C3)
(C1 C2 C4)

?(SAN 'C5)
NIL

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

(SAN *node space*)[1] Set of Adjacent Nodes \'san\

Informal Definition

The function SAN is an EXPR which returns the set of adjacent nodes to *node* in *space*. Given *node* and *space*, SAN returns (m_1 ... m_i ... m_t) where for each i, some edge g_i points to node m_i from *node* in *space* or points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

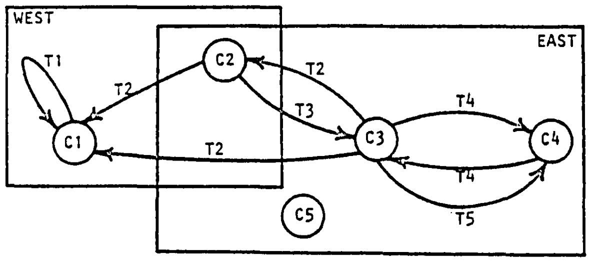
SAN[n,s] = SON[n,s] ∪ SIN[n,s]

error condition:

- ((n s) v) ∉ NSV for all v ∈ V
- s ∉ S

[1]See alternative form on page 152.

Illustrations

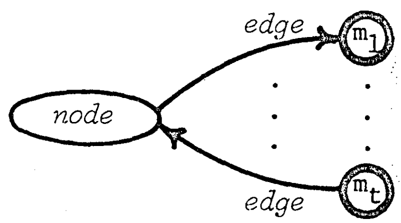



```
?(SAN 'C3 'EAST)
(C2 C4)

?(SAN 'C1 'WEST)
(C1 C2)

?(SAN 'C1 'UNIVERSE)
(C1 C2 C3)

?(SAN 'C5 'EAST)
NIL

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

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

(SANG *node edge*)[1] Set of Adjacent Nodes given an edGe \'saŋ\

Informal Definition

The function SANG is an EXPR which returns the set of adjacent nodes of *node* which are connected by *edge*. Given *node* and *edge*, SANG returns (m_1 ... m_i ... m_t) where for each i, *edge* points to node m_i from *node* or points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

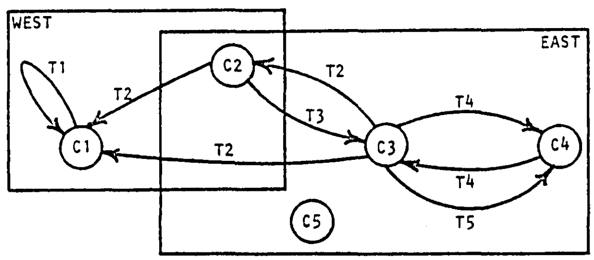
SANG[n,g] = SONG[n,g] ∪ SING[n,g]

error condition:

- $n \notin N$

[1] See alternative form on page 158.

Illustrations

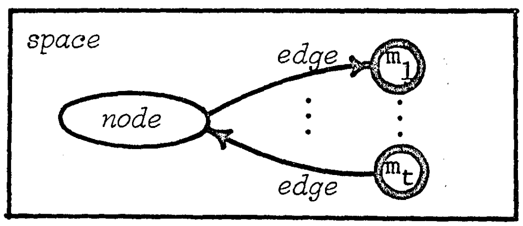



```
?(SANG 'C1 'T2)
(C2 C3)

?(SANG 'C3 'T4)
(C4)

?(SANG 'C1 'T1)
(C1)

?(SANG 'C1 'T4)
NIL

?(SANG 'C5 'T1)
NIL

?(SANG 'C1 'TX)
NIL

?(SANG 'CX 'T1)
*** SANG ERROR: CX IS NOT A NODE
```

(SANG *node edge space*)[1] <u>S</u>et of <u>A</u>djacent <u>N</u>odes given an ed<u>G</u>e \'saŋ\

Informal Definition

The function SANG is an EXPR which returns the set of adjacent nodes of *node* which are connected by *edge* in *space*. Given *node*, *edge*, and *space*, SANG returns (m_1 ... m_i ... m_t) where for each i, *edge* points to node m_i from *node* in *space* or points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

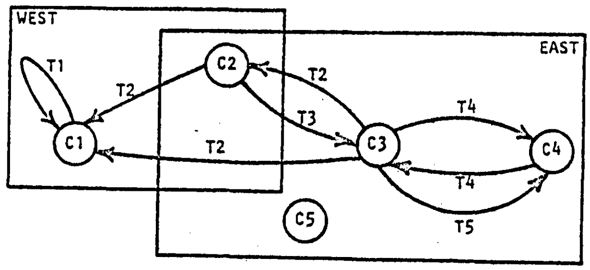
SANG[n,g,s] = SONG[n,g,s] $\cup$ SING[n,g,s]

error conditions:

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

[1] See alternative form on page 156.

Illustrations

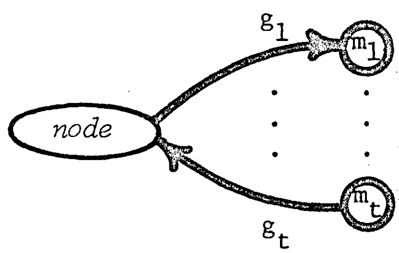



```
?(SANG 'C3 'T4 'EAST)
(C4)

?(SANG 'C1 'T2 'WEST)
(C2)

?(SANG 'C2 'T2 'UNIVERSE)
(C1 C3)

?(SANG 'C3 'TX 'EAST)
NIL

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

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

(SAP *node*)[1] Set of Adjacent Pairs \'sap\

Informal Definition

The function SAP is an EXPR which returns the set of adjacent pairs of *node*. Given *node*, SAP returns $((g_1\ m_1)\ \ldots\ (g_i\ m_i)\ \ldots\ (g_t\ m_t))$ where for each i, edge g_i points to node m_i from *node* or points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

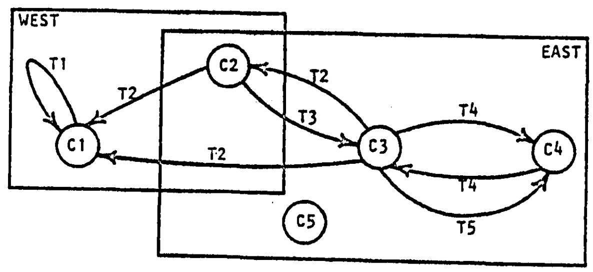
SAP[n] = SOP[n] ∪ SIP[n]

error condition:

- $n \notin N$

[1]See alternative form on page 162.

Illustrations

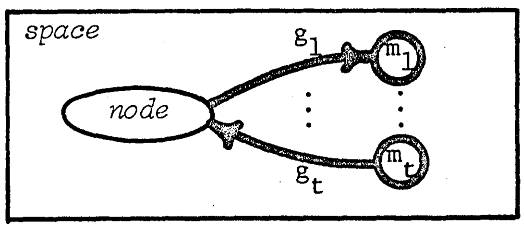



```
?(SAP 'C1)
((T1 C1) (T2 C2) (T2 C3))

?(SAP 'C2)
((T2 C1) (T2 C3) (T3 C3))

?(SAP 'C3)
((T2 C1) (T2 C2) (T3 C2) (T4 C4) (T5 C4))

?(SAP 'C4)
((T4 C3) (T5 C3))

?(SAP 'C5)
NIL

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

(SAP *node space*)[1] Set of Adjacent Pairs \'sap\

Informal Definition

The function SAP is an EXPR which returns the set of adjacent pairs of *node* in *space*. Given *node* and *space*, SAP returns $((g_1\ m_1)\ \ldots\ (g_i\ m_i)\ \ldots\ (g_t\ m_t))$ where for each i, edge g_i points to node m_i from *node* in *space* or points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

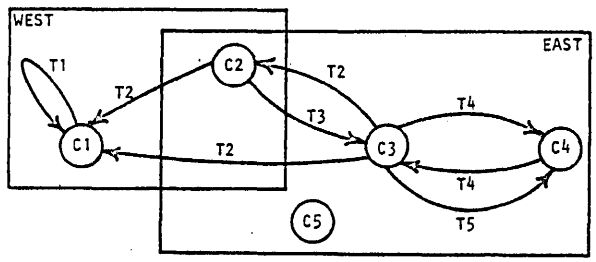
$SAP[n,s] = SOP[n,s] \cup SIP[n,s]$

error conditions:

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

[1]See alternative form on page 160.

Illustrations

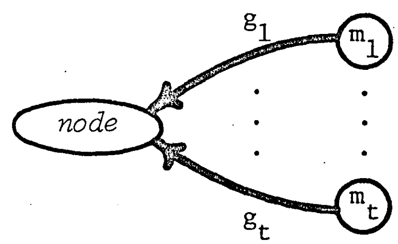



```
?(SAP 'C3 'EAST)
((T2 C2) (T3 C2) (T4 C4) (T5 C4))

?(SAP 'C1 'WEST)
((T1 C1) (T2 C2))

?(SAP 'C1 'UNIVERSE)
((T1 C1) (T2 C2) (T2 C3))

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

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

(SIG *node*)[1] Set of Inpointing edGes \'sig\

Informal Definition

The function SIG is an EXPR which returns the set of inpointing edges of *node*. Given *node*, SIG returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to *node* from some node m_i.



error condition:

- *node* does not exist

Formal Definition

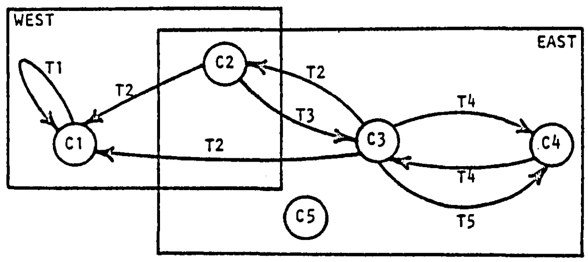
$SIG[n] = \{g \mid \exists m \in N \text{ s.t. } (m\ g\ n) \in NGN\}$

error condition:

- $n \notin N$

[1]See alternative form on page 166.

Illustrations

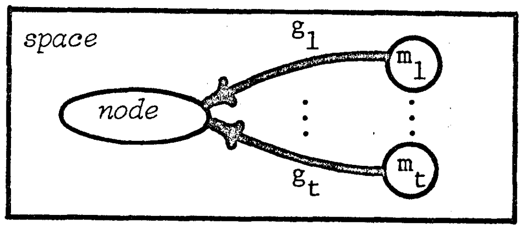



```
?(SIG 'C1)
(T1 T2)

?(SIG 'C2)
(T2)

?(SIG 'C3)
(T3 T4)

?(SIG 'C5)
NIL

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

(SIG *node space*)[1] Set of Inpointing edGes \'sig\

Informal Definition

The function SIG is an EXPR which returns the set of inpointing edges of *node* in *space*. Given *node* and *space*, SIG returns $(g_1 \ldots g_i \ldots g_t)$ where for each i, edge g_i points to *node* from some node m_i in *space*.



error conditions:

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

Formal Definition

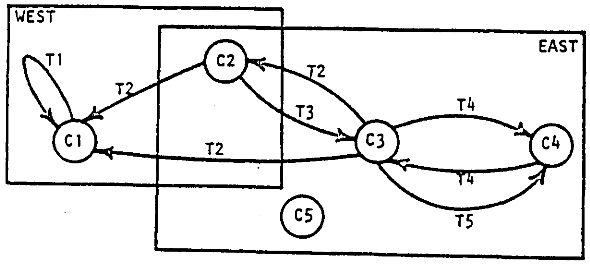
$SIG[n,s] = \{g \mid \exists m \in N, v \in V \text{ s.t. } (((m\ g\ n)\ s)\ v) \in NGNSV\}$

error conditions:

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

[1] See alternative form on page 164.

Illustrations

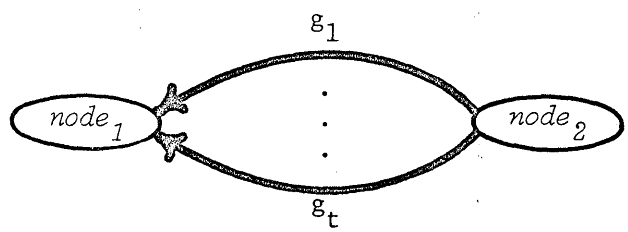



```
?(SIG 'C3 'EAST)
(T3 T4)

?(SIG 'C1 'WEST)
(T1 T2)

?(SIG 'C1 'UNIVERSE)
(T1 T2)

?(SIG 'C5 'EAST)
NIL

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

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

(SIGN $node_1$ $node_2$)[1] Set of Inpointing edGes given a Node \'sig-in\

Informal Definition

The function SIGN is an EXPR which returns the set of inpointing edges of $node_1$ that originate from $node_2$. Given $node_1$ and $node_2$, SIGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to $node_1$ from $node_2$.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

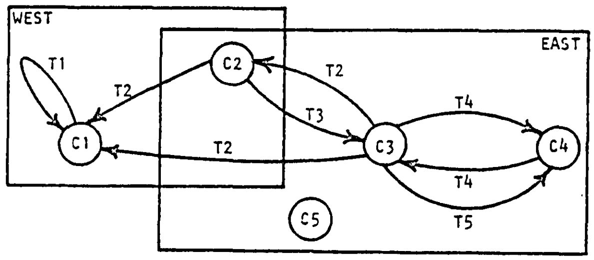
SIGN[n,m] = {g|(m g n) $\in$ NGN}

error conditions:

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

[1]See alternative form on page 170.

Illustrations

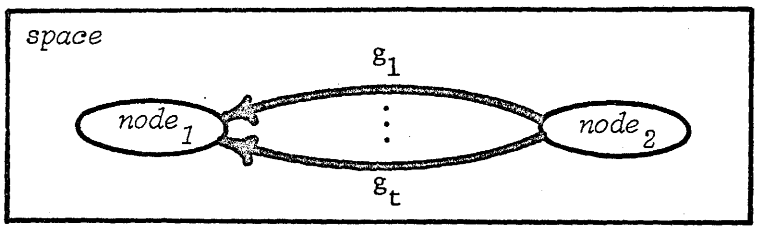



```
?(SIGN 'C1 'C3)
(T2)

?(SIGN 'C4 'C3)
(T4 T5)

?(SIGN 'C1 'C1)
(T1)

?(SIGN 'C1 'CX)
*** SIGN ERROR: CX IS NOT A NODE

?(SIGN 'CX 'C1)
*** SIGN ERROR: CX IS NOT A NODE
```

(SIGN $node_1$ $node_2$ *space*)[1] Set of Inpointing edGes given a Node \'sig-in\

Informal Definition

The function SIGN is an EXPR which returns the set of inpointing edges of $node_1$ that originate from $node_2$ in *space*. Given $node_1$, $node_2$, and *space*, SIGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to $node_1$ from $node_2$ in *space*.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- *space* does not exist

Formal Definition

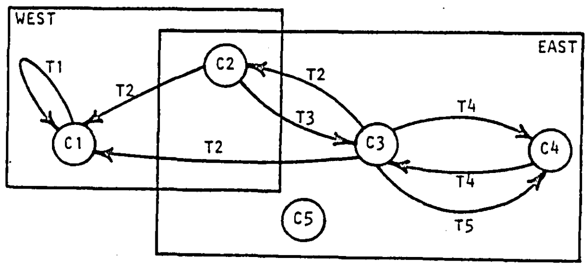
$$SIGN[n,m,s] = \{g \mid \exists v \in V \text{ s.t. } (((m\ g\ n)\ s)\ v) \in NGNSV\}$$

error conditions:

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

[1]See alternative form on page 168.

Illustrations

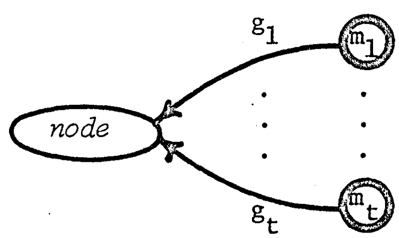



```
?(SIGN 'C4 'C3 'EAST)
(T4 T5)

?(SIGN 'C1 'C1 'WEST)
(T1)

?(SIGN 'C1 'C3 'UNIVERSE)
(T2)

?(SIGN 'C5 'C3 'EAST)
NIL

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

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

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

(SIN *node*)[1] Set of Inpointing Nodes \'sin\

Informal Definition

The function SIN is an EXPR which returns the set of inpointing nodes of *node*. Given *node*, SIN returns (m_1 ... m_i ... m_t) where for each i, some edge g_i points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

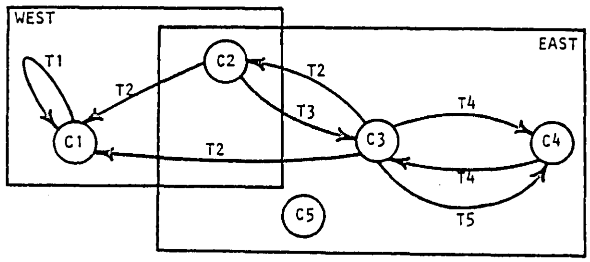
$SIN[n] = \{m | \exists g \in G \text{ s.t. } (m\ g\ n) \in NGN\}$

error condition:

- $n \notin N$

[1]See alternative form on page 174.

Illustrations

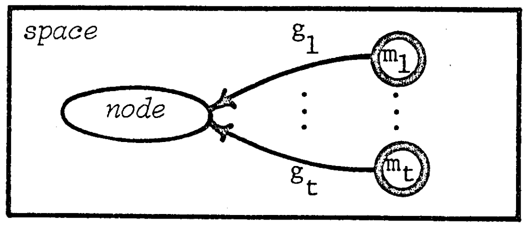



```
?(SIN 'C1)
(C1 C2 C3)

?(SIN 'C3)
(C2 C4)

?(SIN 'C4)
(C3)

?(SIN 'C5)
NIL

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

(SIN *node space*)[1] Set of Inpointing Nodes \'sin\

Informal Definition

The function SIN is an EXPR which returns the set of inpointing nodes of *node* in *space*. Given *node* and *space*, SIN returns (m_1 ... m_i ... m_t) where for each i, some edge g_i points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

SIN[n,s] = {m|∃g ∈ G, v ∈ V s.t. (((m g n) s) v) ∈ NGNSV}

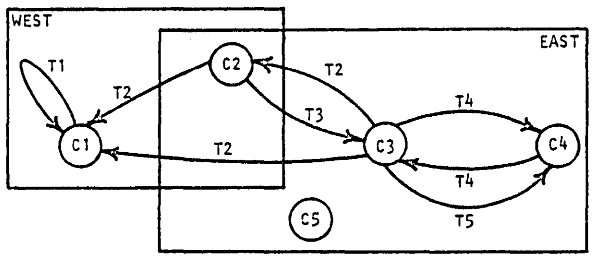
error conditions:

- ((n s) v) ∉ NSV for all v ∈ V
- s ∉ S

[1]See alternative form on page 172.

SIN

Illustrations

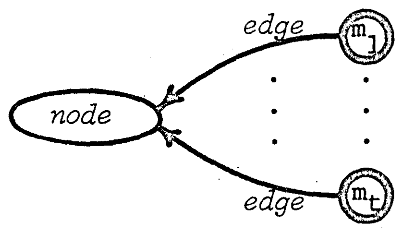



```
?(SIN 'C3 'EAST)
(C2 C4)

?(SIN 'C1 'WEST)
(C1 C2)

?(SIN 'C1 'UNIVERSE)
(C1 C2 C3)

?(SIN 'C5 'EAST)
NIL

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

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

(SING *node edge*)[1] Set of Inpointing Nodes given an edGe \'siŋ\

Informal Definition

The function SING is an EXPR which returns the set of inpointing nodes of *node* along *edge*. Given *node* and *edge*, SING returns (m_1 ... m_i ... m_t) where for each i, *edge* points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

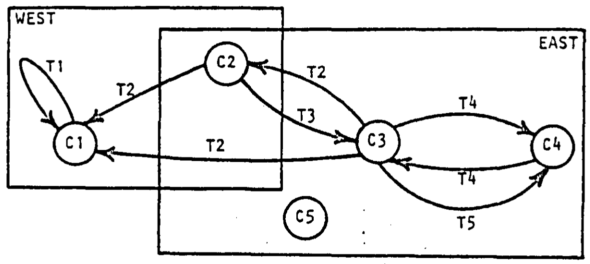
SING[n,g] = {m|(m g n) $\in$ NGN}

error condition:

- n $\notin$ N

[1] See alternative form on page 178.

Illustrations

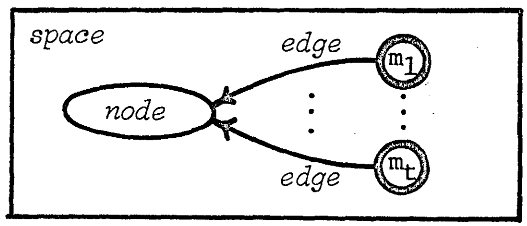



```
?(SING 'C1 'T2)
(C2 C3)

?(SING 'C1 'T1)
(C1)

?(SING 'C4 'T4)
(C3)

?(SING 'C5 'T1)
NIL

?(SING 'C1 'TX)
NIL

?(SING 'CX 'T1)
*** SING ERROR: CX IS NOT A NODE
```

(SING *node edge space*) Set of Inpointing Nodes given an edGe \'siŋ\

Informal Definition

The function SING is an EXPR which returns the set of inpointing nodes of *node* along *edge* in *space*. Given *node*, *edge*, and *space*, SING returns (m_1 ... m_i ... m_t) where for each i, *edge* points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

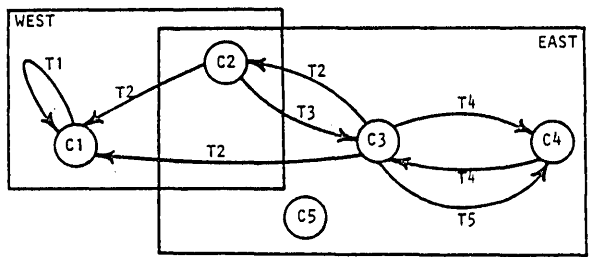
$\text{SING}[n,g,s] = \{m \mid \exists v \in V \text{ s.t. } (((m\ g\ n)\ s)\ v) \in \text{NGNSV}\}$

error conditions:

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

[1]See alternative form on page 176.

Illustrations

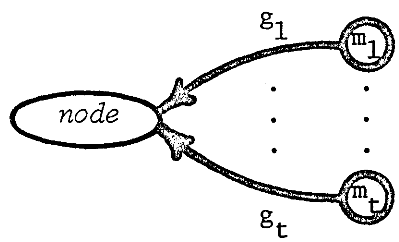



```
?(SING 'C4 'T4 'EAST)
(C3)

?(SING 'C1 'T2 'WEST)
(C2)

?(SING 'C1 'T2 'UNIVERSE)
(C2 C3)

?(SING 'C4 'TX 'EAST)
NIL

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

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

(SIP *node*)[1] Set of Inpointing Pairs \'sip\

Informal Definition

The function SIP is an EXPR which returns the set of inpointing pairs of *node*. Given *node*, SIP returns $((g_1\ m_1)\ \ldots\ (g_i\ m_i)\ \ldots\ (g_t\ m_t))$ where for each i, edge g_i points to *node* from node m_i.



error condition:

- *node* does not exist

Formal Definition

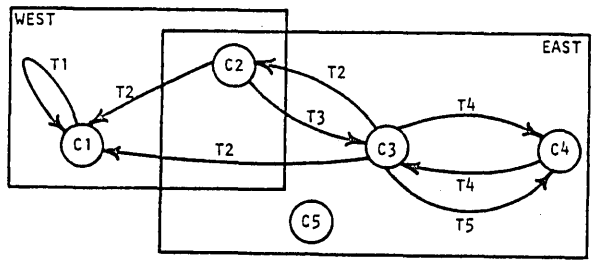
$SIP[n] = \{(g\ m) \mid (m\ g\ n) \in NGN\}$

error condition:

- $n \notin N$

[1]See alternative form on page 182.

Illustrations

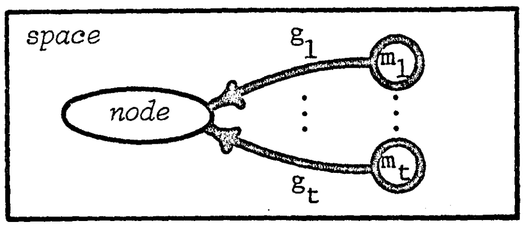



```
?(SIP 'C1)
((T1 C1) (T2 C2) (T2 C3))

?(SIP 'C2)
((T2 C3))

?(SIP 'C5)
NIL

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

(SIP *node space*)[1] Set of Inpointing Pairs \'sip\

Informal Definition

The function SIP is an EXPR which returns the set of inpointing pairs of *node* in *space*. Given *node* and *space*, SIP returns $((g_1\ m_1)\ \ldots\ (g_i\ m_i)\ \ldots\ (g_t\ m_t))$ where for each i, edge g_i points to *node* from node m_i in *space*.



error conditions:

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

Formal Definition

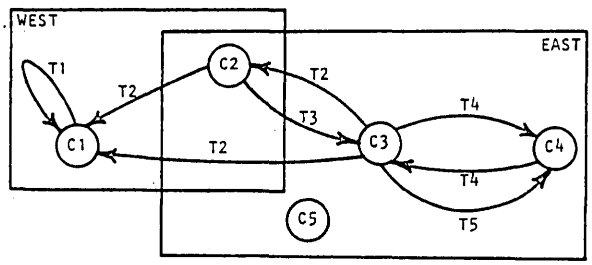
$SIP[n,s] = \{(g\ m) \mid \exists v \in V \text{ s.t. } (((m\ g\ n)\ s)\ v) \in NGNSV\}$

error conditions:

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

[1] See alternative form on page 180.

Illustrations

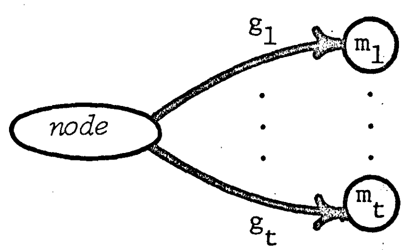



```
?(SIP 'C3 'EAST)
((T3 C2) (T4 C4))

?(SIP 'C1 'WEST)
((T1 C1) (T2 C2))

?(SIP 'C1 'UNIVERSE)
((T1 C1) (T2 C2) (T2 C3))

?(SIP 'C5 'EAST)
NIL

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

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

(SOG *node*)[1] Set of Outpointing edGes \'sȯg\

Informal Definition

The function SOG is an EXPR which returns the set of outpointing edges of *node*. Given *node*, SOG returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to some node m_i from *node*.



error condition:

- *node* does not exist

Formal Definition

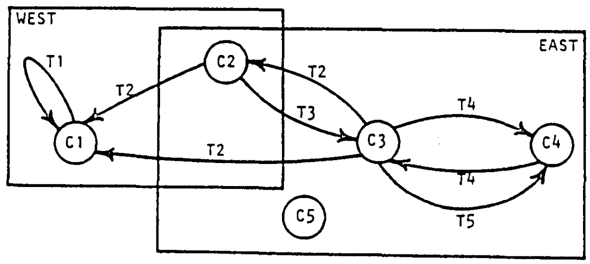
SOG[n] = $\{g \mid \exists m \in N$ s.t. $(n\ g\ m) \in NGN\}$

error condition:

- $n \notin N$

[1]See alternative form on page 186.

Illustrations

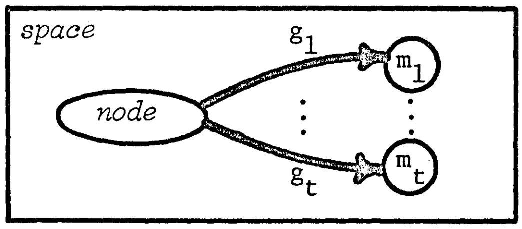



```
?(SOG 'C1)
(T1)

?(SOG 'C3)
(T2 T4 T5)

?(SOG 'C2)
(T2 T3)

?(SOG 'C5)
NIL

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

(SOG *node space*)[1] <u>S</u>et of <u>O</u>utpointing ed<u>G</u>es \'sȯg\

Informal Definition

The function SOG is an EXPR which returns the set of outpointing edges from *node* in *space*. Given *node* and *space*, SOG returns $(g_1 \ldots g_i \ldots g_t)$ where for each i, edge g_i points to some node m_i from *node* in *space*.



error conditions:

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

Formal Definition

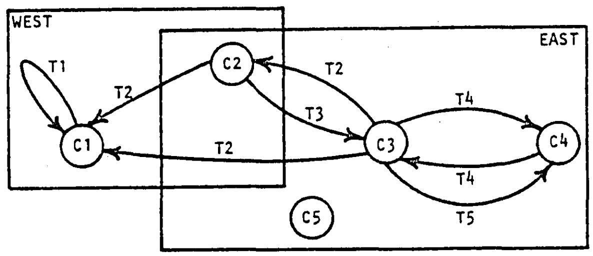
$$SOG[n,s] = \{g \mid \exists m \in N,\ v \in V \text{ s.t. } (((n\ g\ m)\ s)\ v) \in NGNSV\}$$

error conditions:

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

[1]See alternative form on page 184.

Illustrations

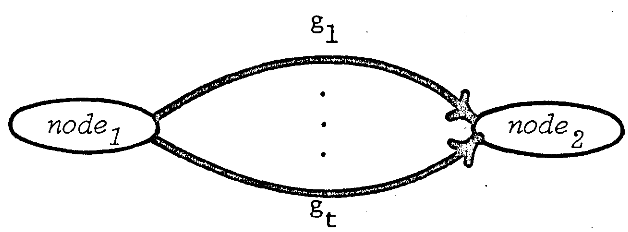



```
?(SOG 'C3 'EAST)
(T2 T4 T5)

?(SOG 'C1 'WEST)
(T1)

?(SOG 'C2 'UNIVERSE)
(T2 T3)

?(SOG 'C5 'EAST)
NIL

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

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

(SOGN *node*$_1$ *node*$_2$)[1] Set of Outpointing edGes given a Node \'sȯg-in\

Informal Definition

The function SOGN is an EXPR which returns the set of outpointing edges of *node*$_1$ that point to *node*$_2$. Given *node*$_1$ and *node*$_2$, SOGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to *node*$_2$ from *node*$_1$.



error conditions:

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

Formal Definition

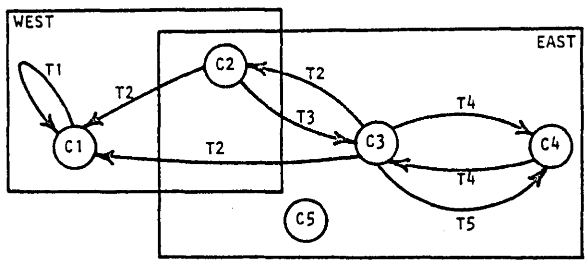
$\text{SOGN}[n,m] = \{g \mid (n\ g\ m) \in \text{NGN}\}$

error conditions:

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

[1]See alternative form on page 190.

Illustrations

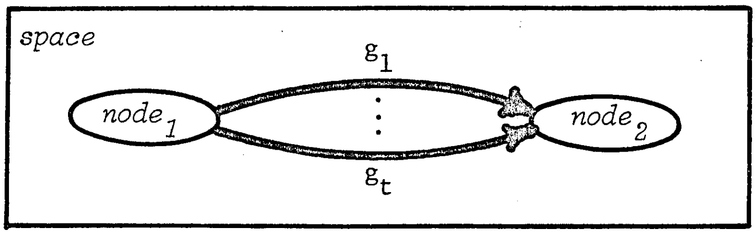



```
?(SOGN 'C2 'C3)
(T3)

?(SOGN 'C3 'C4)
(T4 T5)

?(SOGN 'C3 'C1)
(T2)

?(SOGN 'C5 'C3)
NIL

?(SOGN 'C1 'CX)
*** SOGN ERROR: CX IS NOT A NODE

?(SOGN 'CX 'C1)
*** SOGN ERROR: CX IS NOT A NODE
```

(SOGN $node_1$ $node_2$ *space*)[1] Set of Outpointing edGes given a Node \'sȯg-in\

Informal Definition

The function SOGN is an EXPR which returns the set of outpointing edges of $node_1$ that point to $node_2$ in *space*. Given $node_1$, $node_2$, and *space*, SOGN returns (g_1 ... g_i ... g_t) where for each i, edge g_i points to $node_2$ from $node_1$ in *space*.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- *space* does not exist

Formal Definition

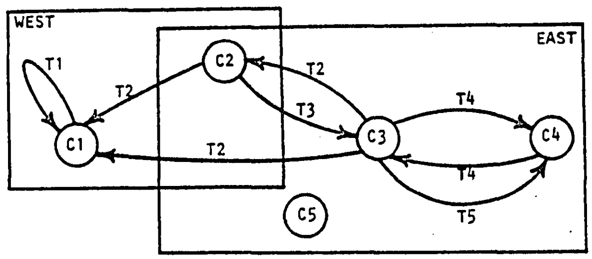
SOGN[n,m,s] = $\{g | \exists v \in V$ s.t. (((n g m) s) v) $\in$ NGNSV$\}$

error conditions:

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

[1]See alternative form on page 188.

Illustrations

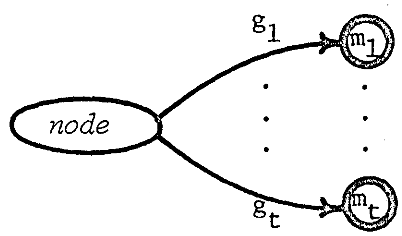



```
?(SOGN 'C3 'C4 'EAST)
(T4 T5)

?(SOGN 'C1 'C1 'WEST)
(T1)

?(SOGN 'C3 'C1 'UNIVERSE)
(T2)

?(SOGN 'C1 'C2 'WEST)
NIL

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

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

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

(SON *node*)[1] Set of Outpointing Nodes \'sän\[2]

Informal Definition

The function SON is an EXPR which returns the set of outpointing nodes of *node*. Given *node*, SON returns (m_1 ... m_i ... m_t) where for each i, some edge g_i points to node m_i from *node*.



error condition:

- *node* does not exist

Formal Definition

$\mathrm{SON}[n] = \{m \mid \exists g \in G \text{ s.t. } (n\ g\ m) \in \mathrm{NGN}\}$

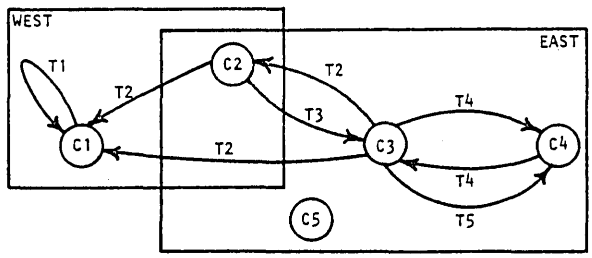
error condition:

- $n \notin N$

[1]See alternative form on page 194.

[2]Note the non-standard pronunciation differing from the pronunciation of SUN.

Illustrations

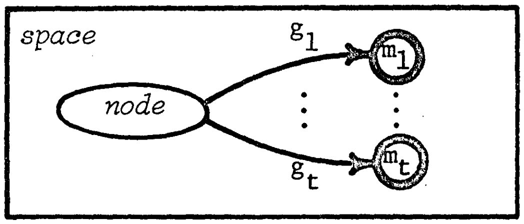



```
?(SON 'C1)
(C1)

?(SON 'C3)
(C1 C2 C4)

?(SON 'C4)
(C3)

?(SON 'C5)
NIL

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

(SON *node space*)[1] Set of Outpointing Nodes \\'sän\\[2]

Informal Definition

The function SON is an EXPR which returns the set of outpointing nodes of *node* in *space*. Given *node* and *space*, SON returns (m_1 ... m_i ... m_t) where for each i, some edge g_i points to node m_i from *node* in *space*.



error conditions:

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

Formal Definition

$SON[n,s] = \{m \mid \exists g \in G,\ v \in V \text{ s.t. } (((n\ g\ m)\ s)\ v) \in NGNSV\}$

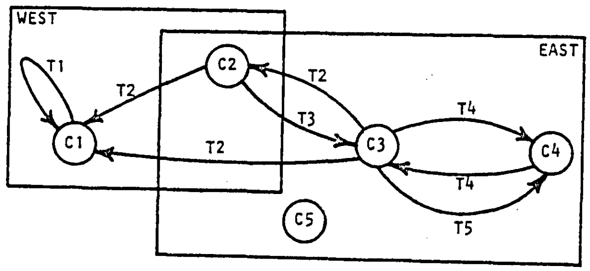
error conditions:

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

[1]See alternative form on page 192.

[2]Note the non-standard pronunciation differing from the pronunciation of SUN.

Illustrations

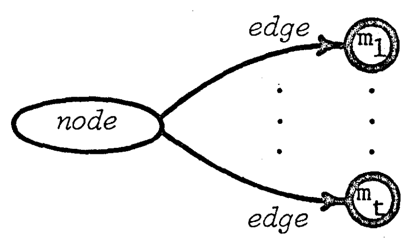



```
?(SON 'C3 'EAST)
(C2 C4)

?(SON 'C1 'WEST)
(C1)

?(SON 'C3 'UNIVERSE)
(C1 C2 C4)

?(SON 'C5 'EAST)
NIL

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

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

(SONG *node edge*)[1] Set of Outpointing Nodes given an edGe \'sȯŋ\

Informal Definition

The function SONG is an EXPR which returns the set of outpointing nodes of *node* which are pointed to by *edge*. Given *node* and *edge*, SONG returns (m_1 ... m_i ... m_t) where for each i, *edge* points to node m_i from *node*.



error condition:

- *node* does not exist

Formal Definition

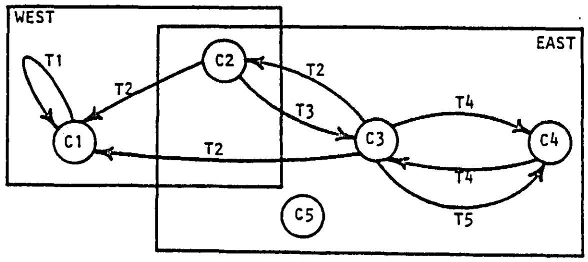
$SONG[n,g] = \{m | (n\ g\ m) \in NGN\}$

error condition:

- $n \notin N$

[1] See alternative form on page 198.

Illustrations

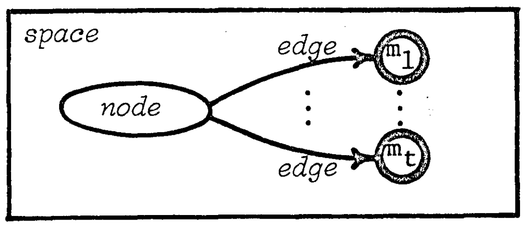



```
?(SONG 'C1 'T1)
(C1)

?(SONG 'C3 'T2)
(C1 C2)

?(SONG 'C5 'T1)
NIL

?(SONG 'C1 'TX)
NIL

?(SONG 'CX 'T1)
*** SONG ERROR: CX IS NOT A NODE
```

(SONG *node edge space*)[1] <u>S</u>et of <u>O</u>utpointing <u>N</u>odes given an ed<u>G</u>e \'sȯŋ\

Informal Definition

The function SONG is an EXPR which returns the set of outpointing nodes of *node* which are pointed to by *edge* in *space*. Given *node*, *edge*, and *space*, SONG returns (m_1 ... m_i ... m_t) where for each i, *edge* points to node m_i from *node* in *space*.



error conditions:

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

Formal Definition

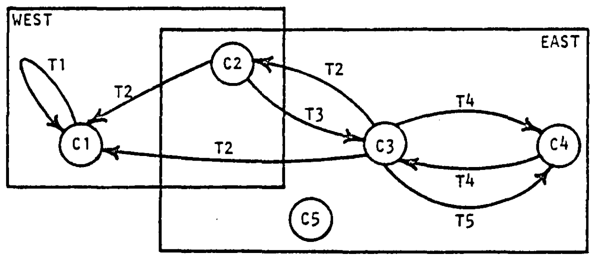
SONG[n,g,s] = $\{m \mid \exists v \in V$ s.t. $(((n\ g\ m)\ s)\ v) \in NGNSV\}$

error conditions:

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

[1]See alternative form on page 196.

Illustrations

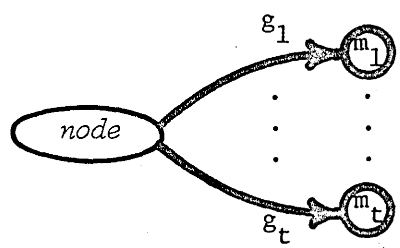



```
?(SONG 'C3 'T2 'EAST)
(C2)

?(SONG 'C1 'T1 'WEST)
(C1)

?(SONG 'C3 'T2 'UNIVERSE)
(C1 C2)

?(SONG 'C3 'TX 'EAST)
NIL

?(SONG 'CX 'T2 'EAST)
*** SONG ERROR: CX IS NOT A NODE IN SPACE EAST

?(SONG 'C3 'T2 'SX)
*** SONG ERROR: SX IS NOT A SPACE
```

(SOP *node*)[1] Set of Outpointing Pairs \'säp\

Informal Definition

The function SOP is an EXPR which returns the set of outpointing pairs of *node*. Given *node*, SOP returns ((g_1 m_1) ... (g_i m_i) ... (g_t m_t)) where for each i, edge g_i points to node m_t from *node*.



error condition:

- *node* does not exist

Formal Definition

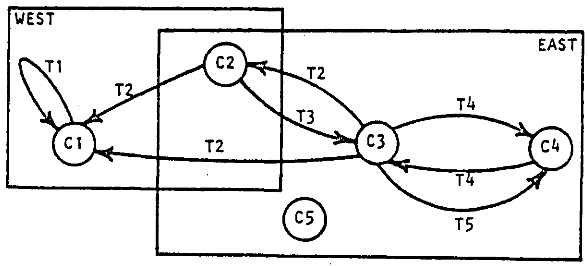
$\mathrm{SOP}[n] = \{(g\ m) \mid (n\ g\ m) \in \mathrm{NGN}\}$

error condition:

- $n \notin N$

[1]See alternative form on page 202.

Illustrations

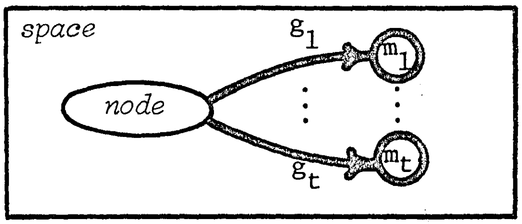



```
?(SOP 'C1)
((T1 C1))

?(SOP 'C3)
((T2 C1) (T2 C2) (T4 C4) (T5 C4))

?(SOP 'C5)
NIL

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

(SOP *node space*)[1] Set of Outpointing Pairs \'säp\

Informal Definition

The function SOP is an EXPR which returns the set of outpointing pairs of *node* in *space*. Given *node* and *space*, SOP returns $((g_1\ m_1) \ldots (g_i\ m_i) \ldots (g_t\ m_t))$ where for each i, edge g_i points to node m_t from *node* in *space*.



error conditions:

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

Formal Definition

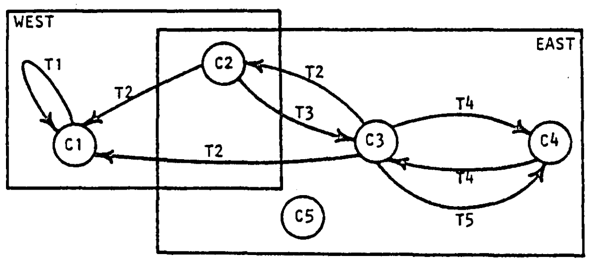
$SOP[n,s] = \{(g\ m) \mid \exists v \in V \text{ s.t. } (((n\ g\ m)\ s)\ v) \in NGNSV\}$

error conditions:

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

[1]See alternative form on page 200.

Illustrations




```
?(SOP 'C3 'EAST)
((T2 C2) (T4 C4) (T5 C4))

?(SOP 'C1 'WEST)
((T1 C1))

?(SOP 'C3 'UNIVERSE)
((T2 C1) (T2 C2) (T4 C4) (T5 C4))

?(SOP 'C5 'EAST)
NIL

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

?(SOP 'C3 'SX)
*** SOP ERROR SX IS NOT A SPACE
```

(SUN)[1] Set of Unqualified Nodes \'sən\

Informal Definition

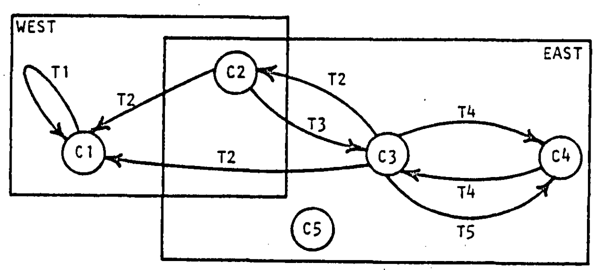
The function SUN is an EXPR which returns the set of existing nodes. SUN returns (m_1 ... m_i ... m_t) where for each i, node m_i exists.

Formal Definition

SUN[ ] = N

[1]See alternative form on page 206.

Illustrations

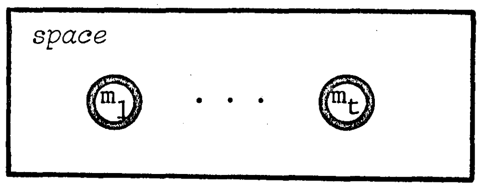



```
?(SUN)
(C1 C2 C3 C4 C5)
```

(SUN *space*)[1] Set of Unqualified Nodes \'sən\

Informal Definition

The function SUN is an EXPR which returns the set of nodes in *space*. Given *space*, SUN returns (m_1 ... m_i ... m_t) where for each i, node m_i exists in *space*.



error condition:

- *space* does not exist

Formal Definition

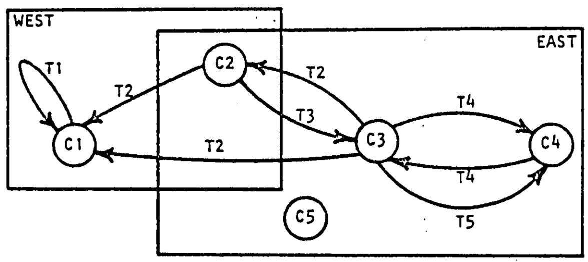
$$SUN[s] = \{n \mid \exists v \in V \text{ s.t. } ((n\ s)\ v) \in NSV\}$$

error condition:

- $s \notin S$

[1]See alternative form on page 204.

Illustrations

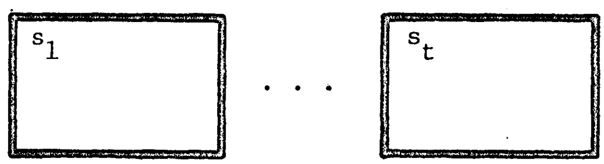



```
?(SUN 'WEST)
(C1 C2)

?(SUN 'EAST)
(C2 C3 C4 C5)

?(SUN 'UNIVERSE)
(C1 C2 C3 C4 C5)

?(SUN 'SX)
*** SUN ERROR: SX IS NOT A SPACE
```

(SUS)[1] Set of Unqualified Spaces \'səs\

Informal Definition

The function SUS is an EXPR which returns the set of existing spaces excluding the universal space. SUS returns (s_1 ... s_i ... s_t) where for each i, s_i is a space other than UNIVERSE.

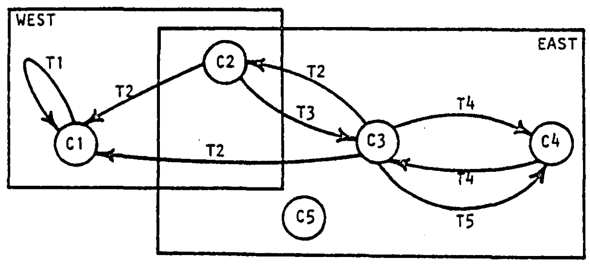


Formal Definition

SUS[ ] = S

[1]See alternative form on page 210.

Illustrations

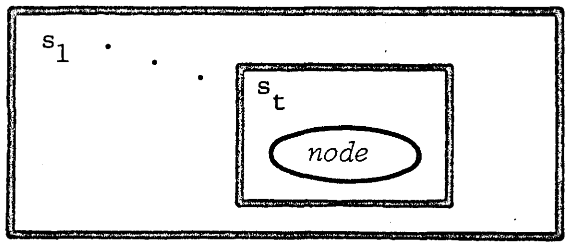



```
?(SUS)
(EAST WEST)
```

(SUS *node*)[1] Set of Unqualified Spaces \'səs\

Informal Definition

The function SUS is an EXPR which returns the set of spaces in which *node* exists excluding the universal space. Given *node*, SUS returns (s_1 ... s_i ... s_t) where for each i, *node* exists in space s_i and s_i is not UNIVERSE.



error condition:

- *node* does not exist

Formal Definition

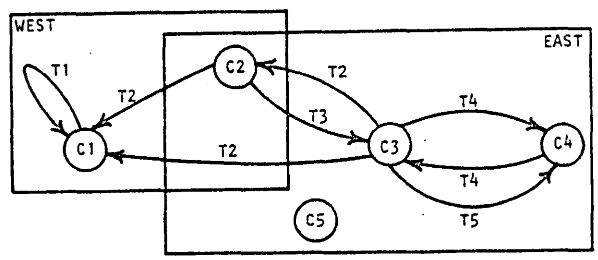
$SUS[n] = \{s \mid \exists v \in V \text{ s.t. } ((n\ s)\ v) \in NSV\} - \{UNIVERSE\}$

error condition:

- $n \notin N$

1 See alternative form on page 208.

Illustrations

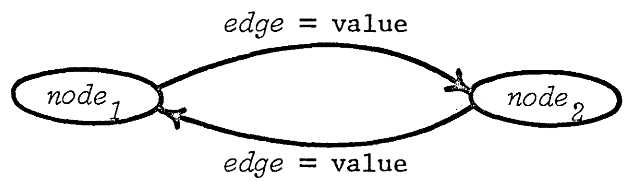



```
?(SUS 'C1)
(WEST)

?(SUS 'C5 )
(EAST)

?(SUS 'C2)
(EAST WEST)

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

(VAP *node*$_1$ *edge* *node*$_2$)[1] Value of Adjacent Pairs \'vap\

Informal Definition

The function VAP is an EXPR which returns the value of the adjacent pairs (*edge* *node*$_2$) of *node*$_1$ in the universal space. Given *node*$_1$, *edge*, and *node*$_2$, VAP returns the value of *edge* pointing from *node*$_1$ to *node*$_2$ and/or pointing from *node*$_2$ to *node*$_1$ in UNIVERSE.



error conditions:

- *node*$_1$ does not exist
- *node*$_2$ does not exist
- (*edge* *node*$_2$) is neither an outpointing nor inpointing pair of *node*$_1$
- the value of the outpointing and inpointing pairs are not equal

Formal Definition

VAP[n,g,m] = v
    where if XOP[n,g,m]
        then v = VOP[n,g,m]
        else v = VIP[n,g,m]

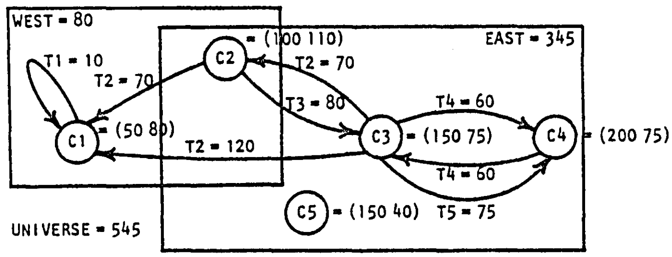
error conditions:

- $n \notin N$
- $m \notin N$
- $(n\ g\ m) \notin NGN$ and $(m\ g\ n) \notin NGN$
- XOP[n,g,m] = T and XIP[n,g,m] = T and VOP[n,g,m] ≠ VIP[n,g,m]

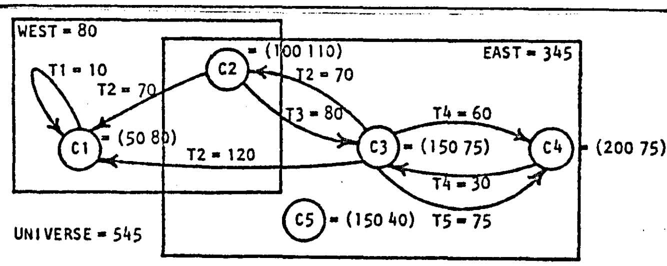
---

[1] See alternative form on page 214.

Illustrations




```
?(VAP 'C3 'T4 'C4)
60

?(VAP 'C4 'T4 'C3)
60

?(VAP 'C1 'T2 'C2)
70

?(VAP 'C2 'T2 'C1)
70

?(VAP 'C3 'T5 'C4)
75

?(VAP 'C3 'TX 'C4)
*** VAP ERROR: THERE IS NO EDGE TX BETWEEN NODE C3 AND NODE C4

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

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

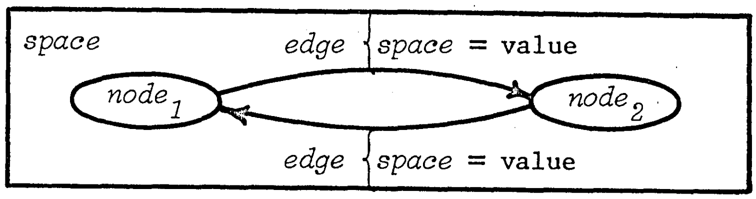



```
?(VAP 'C3 'T4 'C4)
*** VAP ERROR: THE OUTPOINTING AND INPOINTING EDGES T4 BETWEEN NODE C3
               AND NODE C4 DO NOT HAVE EQUAL VALUES
```

(VAP *node*$_1$ *edge* *node*$_2$ *space*) Value of Adjacent Pairs \'vap\

Informal Definition

The function VAP is an EXPR which returns the value of the adjacent pairs (*edge* *node*$_2$) of *node*$_1$ in *space*. Given *node*$_1$, *edge*, *node*$_2$, and *space*, VAP returns the value of *edge* pointing from *node*$_1$ to *node*$_2$ and/or pointing from *node*$_2$ to *node*$_1$ in *space*.



error conditions:

- *node*$_1$ does not exist in *space*
- *node*$_2$ does not exist in *space*
- (*edge* *node*$_2$) is neither an outpointing nor inpointing pair of *node*$_1$ in *space*
- the values of the outpointing and inpointing pairs are not equal
- *space* does not exist

Formal Definition

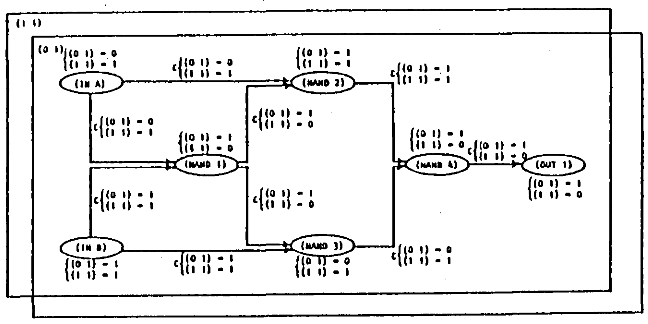
VAP[n,g,m,s] = v
    where if XOP[n,g,m,s] = T
        then v = VOP[n,g,m,s]
        else v = VIP[n,g,m,s]

error conditions:

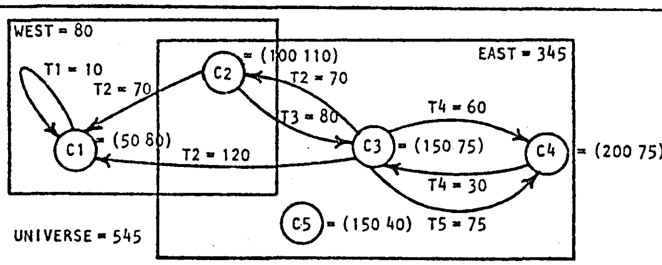
- ((n s) v') $\notin$ NSV for all v' $\in$ V
- ((m s) v') $\notin$ NSV for all v' $\in$ V
- (((n g m) s) v) $\notin$ NGNSV for all v $\in$ V
  and (((m g n) s) v) $\notin$ NGNSV for all v $\in$ V
- XOP[n,g,m,s] = T and XIP[n,g,m,s] = T and VOP[n,g,m,s] $\neq$ VIP[n,g,m,s]
- s $\notin$ S

[1] See alternative form on page 212.

Illustrations



```
?(VAP '(IN A) 'C '(NAND 2) '(0 1))
0

?(VAP '(NAND 1) 'C '(IN A) '(1 1))
1

?(VAP '(NAND 1) 'C '(IN A) '(0 1))
0

?(VAP '(IN A) 'C '(NAND 2) 'UNIVERSE)
NIL
```

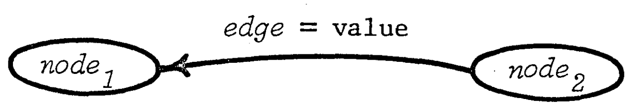



```
?(VAP 'C3 'TX 'C4 'UNIVERSE)
*** VAP ERROR: THERE IS NO EDGE TX BETWEEN NODE C3 AND NODE C4

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

?(VAP 'C3 'T4 'CX 'UNIVERSE)
*** VAP ERROR: CX IS NOT A NODE

?(VAP 'C3 'T4 'C4 'UNIVERSE)
*** VAP ERROR: THE OUTPOINTING AND INPOINTING EDGES T4 BETWEEN NODE C3
               AND NODE C4 DO NOT HAVE EQUAL VALUES

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

(VIP $node_1$ $edge$ $node_2$)[1] Value of Inpointing Pair \'vip\

Informal Definition

The function VIP is an EXPR which returns the value of the inpointing pair ($edge$ $node_2$) of $node_1$ in the universal space. Given $node_1$, $edge$, and $node_2$, VIP returns the value of $edge$ pointing from $node_2$ to $node_1$ in UNIVERSE.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist
- ($edge$ $node_2$) is not an inpointing pair of $node_1$

Formal Definition

VIP[n,g,m] = v

where (((m g n) UNIVERSE) v) $\in$ NGNSV

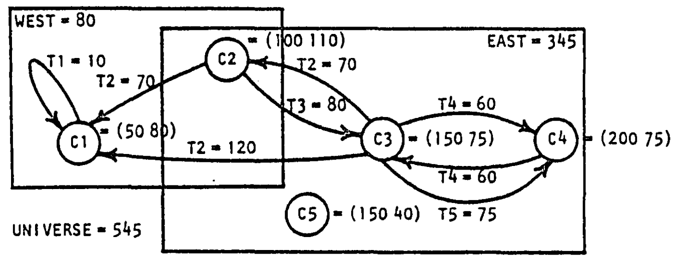
error conditions:

- n $\notin$ N
- m $\notin$ N
- (m g n) $\notin$ NGN

---

[1] See alternative form on page 218.

Illustrations

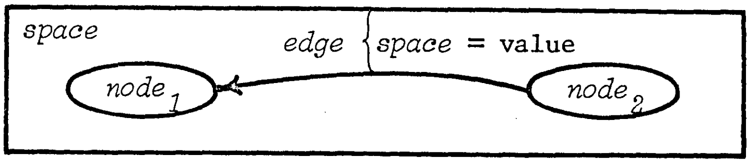



```
?(VIP 'C3 'T4 'C4)
60

?(VIP 'C1 'T2 'C2)
70

?(VIP 'C1 'T1 'C1)
10

?(VIP 'C1 'T3 'C3)
*** VIP ERROR: THERE IS NO EDGE T3 POINTING FROM NODE C1 TO NODE C3

?(VIP 'C1 'TX 'C3)
*** VIP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C1 TO NODE C3

?(VIP 'CX 'T2 'C3)
*** VIP ERROR: CX IS NOT A NODE

?(VIP 'C1 'T2 'CX)
*** VIP ERROR: CX IS NOT A NODE
```

(VIP $node_1$ $edge$ $node_2$ $space$) Value of Inpointing Pair \'vip\

Informal Definition

The function VIP is an EXPR which returns the value of the inpointing pair ($edge$ $node_2$) of $node_1$ in $space$. Given $node_1$, $edge$, $node_2$, and $space$, VIP returns the value of $edge$ pointing from $node_2$ to $node_1$ in $space$.



error conditions:

- $node_1$ does not exist in $space$
- $node_2$ does not exist in $space$
- ($edge$ $node_2$) is not an inpointing pair of $node_1$ in $space$
- $space$ does not exist

Formal Definition

VIP[n,g,m,s] = v

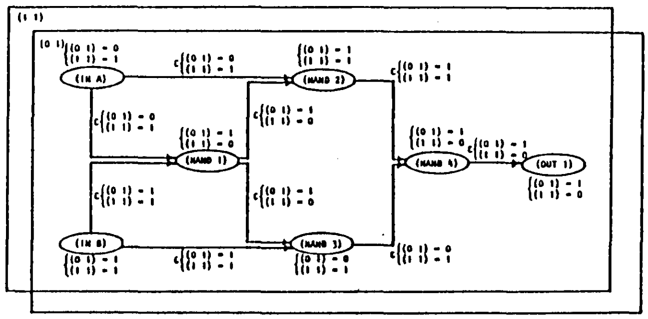
where $(((m\ g\ n)\ s\ v) \in NGNSV$

error conditions:

- $((n\ s)\ v') \notin NSV$ for all $v' \in V$
- $((m\ s)\ v') \notin NSV$ for all $v' \in V$
- $(((m\ g\ n)\ s)\ v) \notin NGNSV$ for all $v \in V$
- $s \notin S$

[1] See alternative form on page 216.

Illustrations

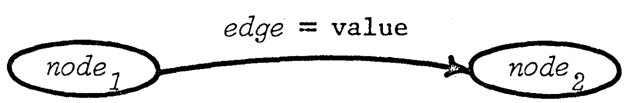



```
?(VIP '(NAND 1) 'C '(IN A) '(1 1))
1

?(VIP '(NAND 1) 'C '(IN A) '(0 1))
0

?(VIP '(NAND 2) 'C '(NAND 1) 'UNIVERSE)
NIL

?(VIP '(NAND 1) 'GX '(IN A) '(1 1))
*** VIP ERROR: THERE IS NO EDGE GX POINTING FROM NODE (NAND 1) TO NODE
               (IN A) IN SPACE (1 1)

?(VIP '(NAND X) 'C '(IN A) '(1 1))
*** VIP ERROR: (NAND X) IS NOT A NODE IN SPACE (1 1)

?(VIP '(NAND 1) 'C '(IN X) '(1 1))
*** VIP ERROR: (IN X) IS NOT A NODE IN SPACE (1 1)

?(VIP '(NAND 1) 'C '(IN A) 'SX)
*** VIP ERROR: SX IS NOT A SPACE
```

(VOP $node_1$ $edge$ $node_2$)[1] Value of Outpointing Pair \'väp\

Informal Definition

The function VOP is an EXPR which returns the value of the outpointing pair ($edge$ $node_2$) of $node_1$ in the universal space. Given $node_1$, $edge$, and $node_2$, VOP returns the value of $edge$ pointing from $node_1$ to $node_2$ in UNIVERSE.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist
- ($edge$ $node_2$) is not an outpointing pair of $node_1$

Formal Definition

VOP[n,g,m] = v

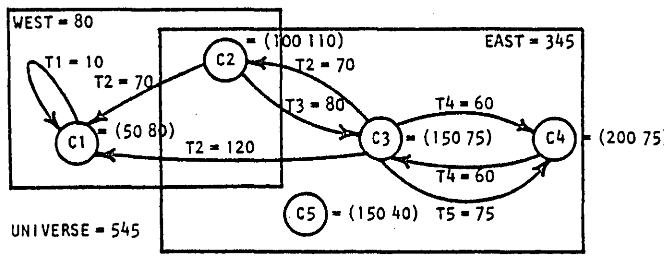
where (((n g m) UNIVERSE) v) ϵ NGNSV

error conditions:

- n $\notin$ N
- m $\notin$ N
- (n g m) $\notin$ NGN

[1]See alternative form on page 222.

Illustrations

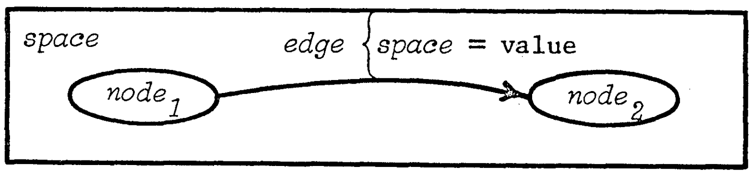



```
?(VOP 'C3 'T4 'C4)
60

?(VOP 'C3 'T2 'C2)
70

?(VOP 'C1 'T1 'C1)
10

?(VOP 'C1 'T2 'C3)
*** VOP ERROR: THERE IS NO EDGE T2 POINTING FROM NODE C1 TO NODE C3

?(VOP 'C1 'TX 'C3)
*** VOP ERROR: THERE IS NO EDGE TX POINTING FROM NODE C1 TO NODE C3

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

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

(VOP $node_1$ *edge* $node_2$ *space*)[1] Value of Outpointing Pair \'väp\

Informal Definition

The function VOP is an EXPR which returns the value of the outpointing pair (*edge* $node_2$) of $node_1$ in *space*. Given $node_1$, *edge*, $node_2$, and *space*, VOP returns the value of *edge* pointing from $node_1$ to $node_2$ in *space*.



error conditions:

- $node_1$ does not exist in *space*
- $node_2$ does not exist in *space*
- (*edge* $node_2$) is not an outpointing pair of $node_1$ in *space*
- *space* does not exist

Formal Definition

VOP[n,g,m,s] = v

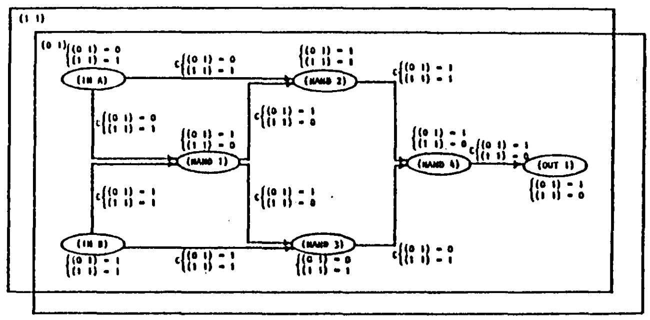
where $(((n\ g\ m)\ s)\ v) \in$ NGNSV

error conditions:

- $((n\ s)\ v') \notin$ NSV for all $v' \in V$
- $((m\ s)\ v') \notin$ NSV for all $v' \in V$
- $(((n\ g\ m)\ s)\ v) \notin$ NGNSV for all $v \in V$
- $s \notin S$

[1]See alternative form on page 220.

Illustrations



```
?(VOP '(IN A) 'C '(NAND 2) '(0 1))
0

?(VOP '(IN A) 'C '(NAND 2) '(1 1))
1

?(VOP '(IN B) 'C '(NAND 1) 'UNIVERSE)
NIL

?(VOP '(IN A) 'GX '(NAND 2) '(0 1))
*** VOP ERROR: THERE IS NO EDGE GX POINTING FROM NODE (IN A) TO NODE
               (NAND 2) IN SPACE (0 1)

?(VOP '(IN X) 'C '(NAND 2) '(0 1))
*** VOP ERROR: (IN X) IS NOT A NODE IN SPACE (0 1)

?(VOP '(IN A) 'C '(NAND X) '(0 1))
*** VOP ERROR: (NAND X) IS NOT A NODE IN SPACE (0 1)

?(VOP '(IN A) 'C '(NAND 2) 'SX)
*** VOP ERROR: SX IS NOT A SPACE
```

(VUN *node*)[1] Value of Unqualified Node \'vən\

Informal Definition

The function VUN is an EXPR which returns the value of *node* in the universal space. Given *node*, VUN returns the value of *node* in UNIVERSE.

node = value

error condition:

- *node* does not exist

Formal Definition

VUN[n] = v

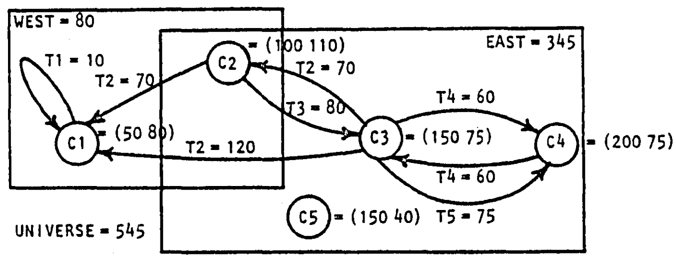
where ((n UNIVERSE) v) $\in$ NSV

error condition:

- n $\notin$ N

[1] See alternative form on page 226.

Illustrations

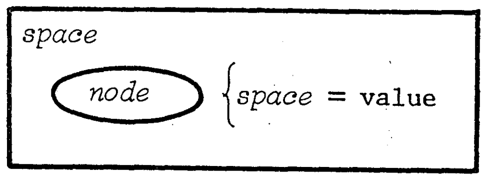



```
?(VUN 'C2)
(100 110)

?(VUN 'C4)
(200 75)

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

(VUN *node space*)[1] Value of Unqualified Node \'vən\

Informal Definition

The function VUN is an EXPR which returns the value of *node* in *space*. Given *node* and *space*, VUN returns the value of *node* in *space*.



error conditions:

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

Formal Definition

VUN[n,s] = v

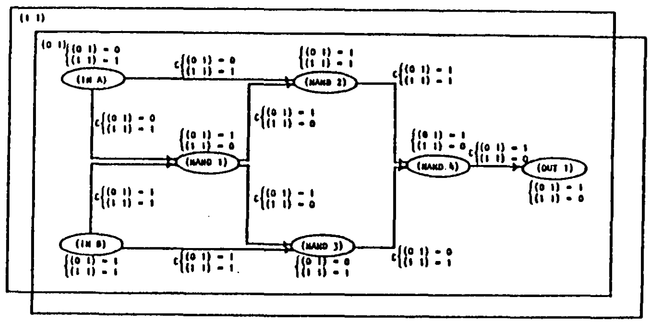
where ((n s) v) $\in$ NSV

error conditions:

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

[1] See alternative form on page 224.

Illustrations

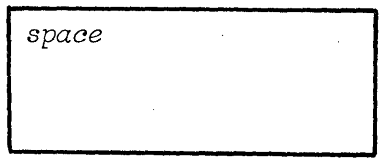


```
?(VUN '(IN A) '(0 1))
0

?(VUN '(IN A) '(1 1))
1

?(VUN '(OUT 1) '(0 1))
1

?(VUN '(OUT 1) 'UNIVERSE)
NIL

?(VUN '(OUT X) '(0 1))
*** VUN ERROR: (OUT X) IS NOT A NODE IN SPACE (0 1)

?(VUN '(OUT 1) 'SX)
*** VUN ERROR: SX IS NOT A SPACE
```

(VUS *space*) Value of Unqualified Space \'vəs\

Informal Definition

The function VUS is an EXPR which returns the value of *space*. Given *space*, VUS returns its value.



error condition:

- *space* does not exist

Formal Definition

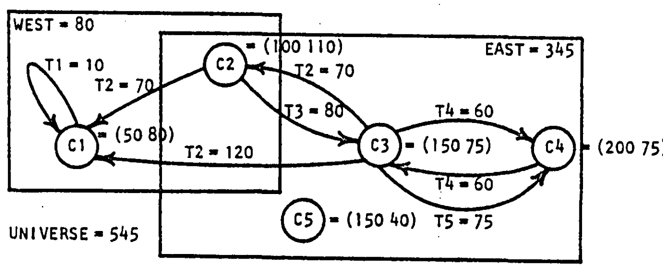
$VUS[s] = v$

where $(s\ v) \in SV$

error condition:

- $s \notin S$

Illustrations

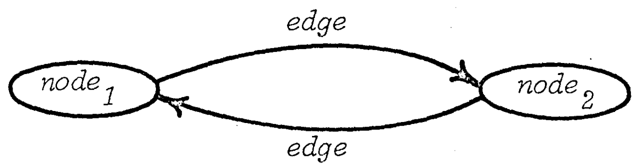



```
?(VUS 'WEST)
80

?(VUS 'EAST)
345

?(VUS 'UNIVERSE)
545

?(VUS 'SX)
*** VUS ERROR: SX IS NOT A SPACE
```

(XAP *node*$_1$ *edge* *node*$_2$)[1] eXistence of Adjacent Pairs \'zap\

Informal Definition

The function XAP is an EXPR which tests for the existence of the adjacent pairs (*edge* *node*$_2$) of *node*$_1$. Given *node*$_1$, *edge*, and *node*$_2$, XAP returns T if *edge* points from *node*$_1$ to *node*$_2$ or from *node*$_2$ to *node*$_1$. Otherwise XAP returns NIL.



error conditions:

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

Formal Definition

XAP[n,g,m] = x

where if XOP[n,g,m] = T or XIP[n,g,n] = T

then x = T

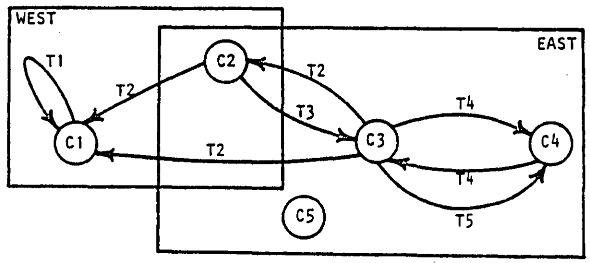
else x = NIL

error conditions:

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

[1]See alternative form on page 232.

Illustrations

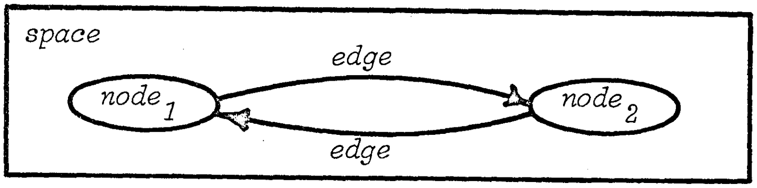



```
?(XAP 'C3 'T4 'C4)
T

?(XAP 'C4 'T4 'C3)
T

?(XAP 'C1 'T1 'C1)
T

?(XAP 'C3 'T2 'C2)
T

?(XAP 'C1 'T4 'C2)
NIL

?(XAP 'C3 'TX 'C4)
NIL

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

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

(XAP $node_1$ $edge$ $node_2$ $space$) eXistence of Adjacent Pairs \'zap\

Informal Definition

The function XAP is an EXPR which tests for the existence of the adjacent pairs ($edge$ $node_2$) of $node_1$ in $space$. Given $node_1$, $edge$, $node_2$, and $space$, XAP returns T if $edge$ points both from $node_1$ to $node_2$ and/or from $node_2$ to $node_1$ in $space$. Otherwise XAP returns NIL.



error conditions:

- $node_1$ does not exist in $space$
- $node_2$ does not exist in $space$
- $space$ does not exist

Formal Definition

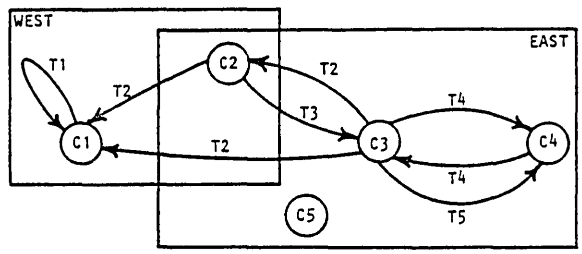
XAP[n,g,m,s] = x
  where if XOP[n,g,m,s] = T or XIP[n,g,m,s] = T
    then x = T
    else x = NIL

error conditions:

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

[1] See alternative form on page 230.

Illustrations

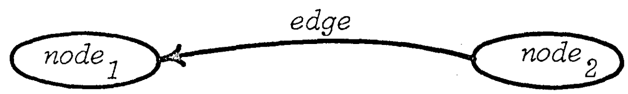



```
?(XAP 'C3 'T4 'C4 'EAST)
T

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

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

?(XAP 'C2 'T4 'C3 'EAST)
NIL

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

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

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

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

(XIP $node_1$ $edge$ $node_2$)[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 $node_1$. 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

Formal Definition

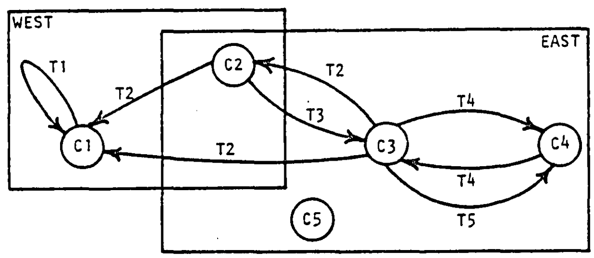
XIP[n,g,m] = x
    where if (m g n) $\in$ NGN
        then x = T
        else x = NIL

error conditions:

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

[1] See alternative form on page 236.

Illustrations

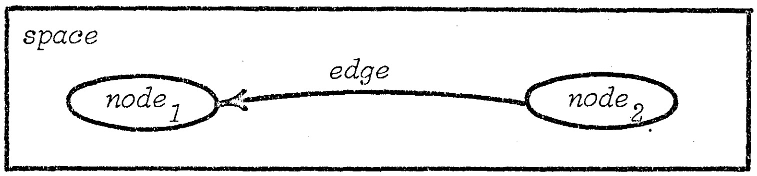



```
?(XIP 'C3 'T4 'C4)
T

?(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
```

(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

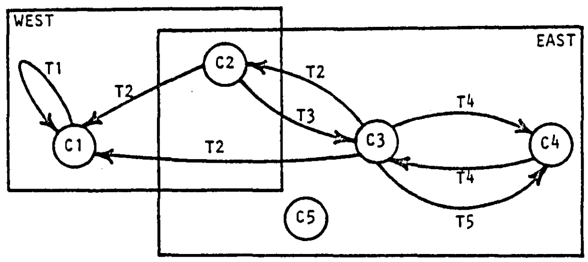
XIP[n,g,m,s] = x
  where if (((m g n) s) v') $\in$ NGNSV for some v' $\in$ V
    then x = T
    else x = NIL

error conditions:

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

---
[1]See alternative form on page 234.

Illustrations

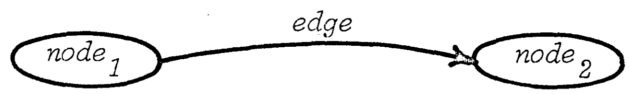



```
?(XIP 'C3 'T4 'C4 'EAST)
T

?(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
```

(XOP $node_1$ $edge$ $node_2$)[1] eXistence of Outpointing Pair \'zäp\

Informal Definition

The function XOP is an EXPR which tests for the existence of the outpointing pair ($edge$ $node_2$) of $node_1$. Given $node_1$, $edge$, and $node_2$, XOP returns T if $edge$ points from $node_1$ to $node_2$ and NIL if it does not.



error conditions:

- $node_1$ does not exist
- $node_2$ does not exist

Formal Definition

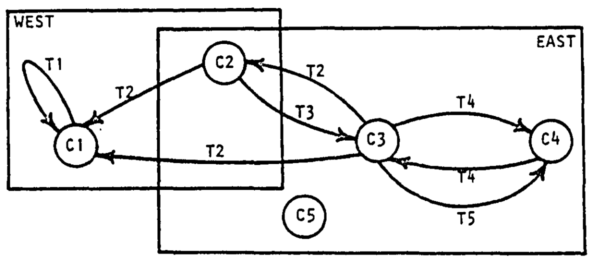
XOP[n,g,m] = x
where if (n g m) ϵ NGN
then x = T
else x = NIL

error conditions:

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

[1]See alternative form on page 240.

Illustrations

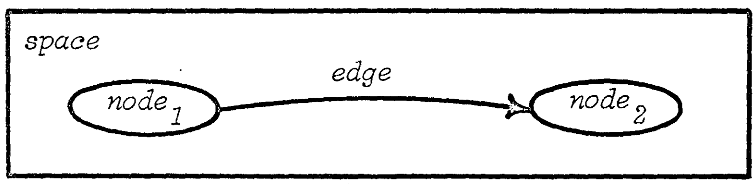



```
?(XOP 'C3 'T4 'C4)
T

?(XOP 'C3 'T2 'C2)
T

?(XOP 'C1 'T1 'C1)
T

?(XOP 'C1 'T2 'C2)
NIL

?(XOP 'C3 'TX 'C4)
NIL

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

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

(XOP $node_1$ *edge* $node_2$ *space*)[1] eXistence of Outpointing Pair \'zäp\

Informal Definition

The function XOP is an EXPR which tests for the existence of the outpointing pair (*edge* $node_2$) of $node_1$ in *space*. Given $node_1$, *edge*, $node_2$, and *space*, XUN returns T if *edge* points from $node_1$ to $node_2$ 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

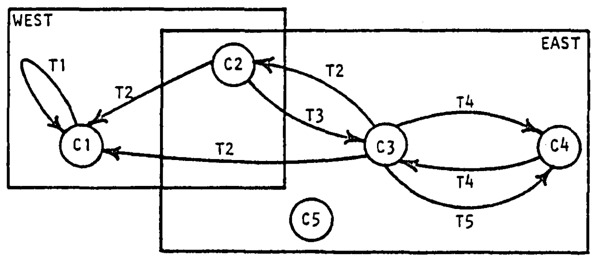
XOP[n,g,m,s] = x
  where if (((n g m) s) v') $\in$ NGNSV for some v' $\in$ V
    then x = T
    else x = NIL

error conditions:

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

[1]See alternative form on page 238.

Illustrations

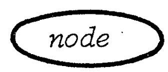



```
?(XOP 'C3 'T4 'C4 'EAST)
T

?(XOP 'C1 'T1 'C1 'WEST)
T

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

?(XOP 'C1 'T2 'C2 'WEST)
NIL

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

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

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

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

(XUN *node*)[1] eXistence of Unqualified Node \'zən\

Informal Definition

The function XUN is an EXPR which tests for the existence of *node*. Given *node*, XUN returns T if *node* exists and NIL if it does not.

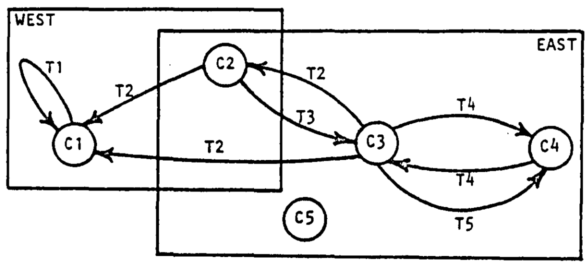


Formal Definition

XUN[n] = x
  where if $n \in N$
    then x = T
    else x = NIL

[1] See alternative form on page 244.

Illustrations

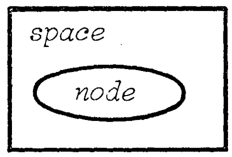



```
?(XUN 'C4)
T

?(XUN 'C2)
T

?(XUN 'C5)
T

?(XUN 'CX)
NIL
```

(XUN *node space*)[1] eXistence of Unqualified Node \\'zən\\

Informal Definition

The function XUN is an EXPR which tests for the existence of *node* in *space*. Given *node* and *space*, XUN returns T if *node* exists in *space* and NIL if it does not.



error condition:

- *space* does not exist

Formal Definition

XUN[n,s] = x

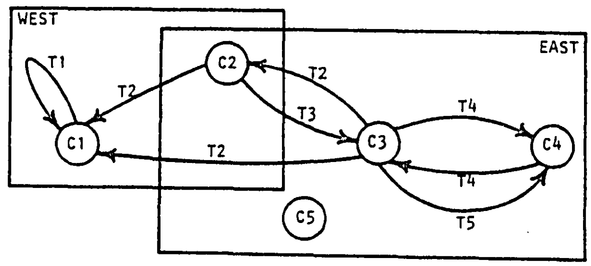
where if (n s) $\in$ NS
then x = T
else x = NIL

error conditions:

- s $\notin$ S

[1] See alternative form on page 242.

Illustrations

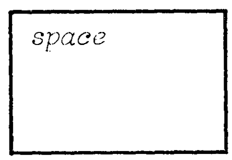



```
?(XUN 'C2 'EAST)
T

?(XUN 'C1 'WEST)
T

?(XUN 'C2 'UNIVERSE)
T

?(XUN 'C4 'UNIVERSE)
T

?(XUN 'C1 'EAST)
NIL

?(XUN 'C4 'WEST)
NIL

?(XUN 'CX 'EAST)
NIL

?(XUN 'C2 'SX)
*** XUN ERROR: SX IS NOT A SPACE
```

(XUS *space*) eXistence of Unqualified Space \'zəs\

Informal Definition

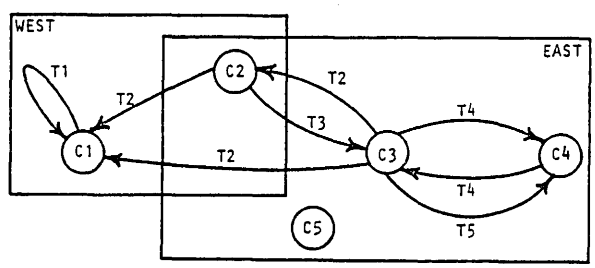
The function XUS is an EXPR which tests for the existence of *space*. Given *space*, XUS returns T if *space* exists and NIL if it does not.



Formal Definition

XUS[s] = x
  where if s ∊ S
      then x = T
      else x = NIL

Illustrations




```
?(XUS 'WEST)
T

?(XUS 'EAST)
T

?(XUS 'UNIVERSE)
T

?(XUS 'SX)
NIL
```

Group II Primitives

Group II primitives involve the destruction, creation, and description of major portions of GRASPER-GRAPHs. In the following two sections descriptors, S-expressions which describe portions of GRAPHs, and Group II switches are described. Most Group II operators are either passed a descriptor as an argument or generate one. Switches inhibit certain aspects of Group II operators. An understanding of descriptors and switches is a prerequisite to understanding the Group II operators.

The names of Group II operators are composed from more primitive GRASPER concepts. This composition is explained in the third section. The final section contains a complete description of each composable operator. The reader is encouraged to refer to these complete descriptions while reading the rules of operator composition.

Descriptors

Descriptors are S-expressions that describe portions of GRASPER-GRAPHs. GRAPH-DESCRIPTORs and NODE-DESCRIPTORs are used by Group II operators. Their definitions, along with some supporting definitions, follow.

A GRAPH-DESCRIPTOR is an S-expression which describes a complete GRASPER-GRAPH. It consists of a list containing a list of space-value-descriptors (svd) followed by an arbitrary number of NODE-DESCRIPTORs.

GRAPH-DESCRIPTOR = $((svd_1 \ldots svd_t)$
$\quad$ NODE-DESCRIPTOR$_1$ ... NODE-DESCRIPTOR$_u$)

Each space-value-descriptor (svd) in a GRAPH-DESCRIPTOR identifies a space and its value. A space-value-descriptor consists of either a space or a space followed by an equal sign and its value. When a space-value-descriptor does not include a value, it is assumed to be NIL. Since the universal space exists in all GRAPHS, it only needs to be included if it has a value other than NIL.

svd = space|
$\quad$ space = value

A NODE-DESCRIPTOR is an S-expression which describes a node, the spaces in which it is included, its value in each of those spaces, and its outpointing and inpointing pairs with their associated spaces and values. A NODE-DESCRIPTOR consists of a list containing a node followed by a list of node-space-value-descriptors (nsvd), a list of outpointing-pair-descriptors (opd), and a list of inpointing-pair-descriptors (ipd). Null lists at the end of a NODE-DESCRIPTOR need not be included.

$$
\begin{aligned}
\text{NODE-DESCRIPTOR} = \; & (\text{node}) \mid \\
& (\text{node } (\text{nsvd}_1 \ldots \text{nsvd}_t)) \mid \\
& (\text{node } (\text{nsvd}_1 \ldots \text{nsvd}_t)(\text{opd}_1 \ldots \text{opd}_u)) \mid \\
& (\text{node } (\text{nsvd}_1 \ldots \text{nsvd}_t) \\
& \quad (\text{opd}_1 \ldots \text{opd}_u) \\
& \quad (\text{ipd}_1 \ldots \text{ipd}_w))
\end{aligned}
$$

Each node-space-value-descriptor (nsvd) indicates the existence and the value of the node from the NODE-DESCRIPTOR in a particular space. Each node-space-value-descriptor consists of either a space or a space followed by an equal sign and the node's value in that space. When a node-space-value-descriptor does not include a value, it is assumed to be NIL. Since all nodes exist in the universal space, a node-space-value-descriptor for UNIVERSE need only be included in a NODE-DESCRIPTOR if the node has a value other than NIL in UNIVERSE.

$$
\begin{aligned}
\text{nsvd} = \; & \text{space} \mid \\
& \text{space} = \text{value}
\end{aligned}
$$

Each outpointing-pair-descriptor (opd) identifies an outpointing pair from the node in the NODE-DESCRIPTOR, the spaces in which it is included, and its value in each of those spaces. An outpointing-pair-descriptor consists of a list containing the edge followed by the node of the pair and a list of pair-space-value-descriptors (PSVD). If the list of pair-space-value-descriptors is null, it need not be included.

opd = (edge node) |
(edge node ($psvd_1$ ... $psvd_t$))

Each inpointing-pair-descriptor (ipd) identifies an inpointing pair to the node in the NODE-DESCRIPTOR, the spaces in which it is included, and its value in each of those spaces. An inpointing-pair-descriptor consists of a list containing the edge followed by the node of the pair and a list of pair-space-value-descriptors (PSVD). If the list of pair-space-value-descriptors is null, it need not be included.

ipd = (edge node) |
(edge node ($psvd_1$ ... $psvd_t$))

Each pair-space-value-descriptor (psvd) indicates the existence and the value of the edge from the outpointing- or inpointing-pair-descriptor in a particular space. Each pair-space-value-descriptor consists of either a space, or a space followed by an equal sign and the edge's value in that space. When a pair-space-value-descriptor does not include a value, it is assumed to be NIL. Since all edges exist in the universal space, a pair-space-value-descriptor for UNIVERSE need only be included in an outpointing- or inpointing-pair-descriptor if it has a value other than NIL in UNIVERSE.

psvd = space |
space = value

space ϵ {S-expressions representing spaces}

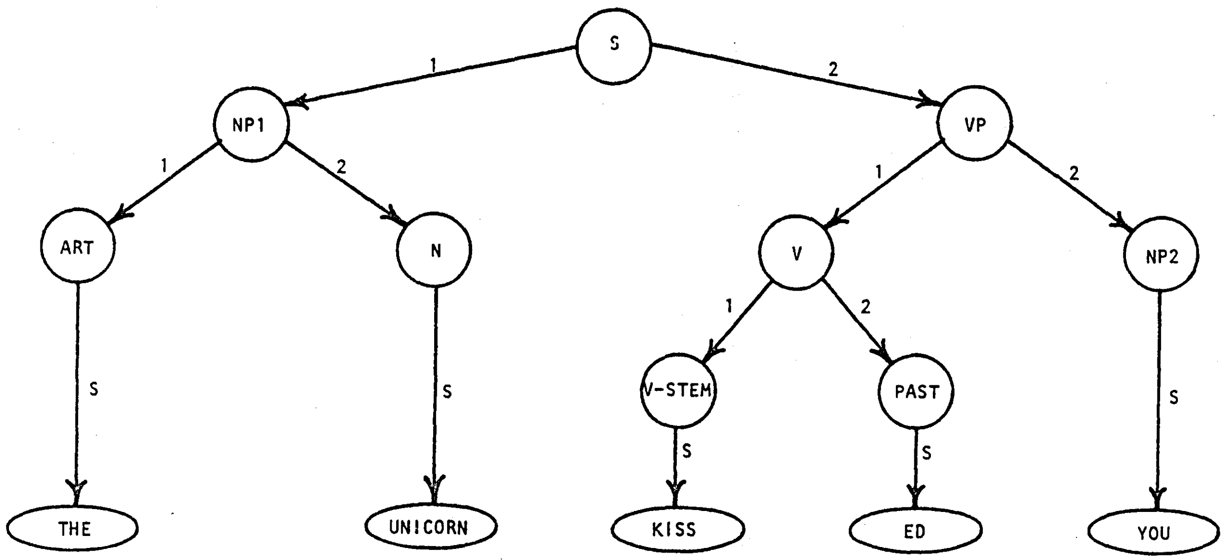
node ϵ {S-expressions representing nodes}

edge ϵ {S-expressions representing edges}

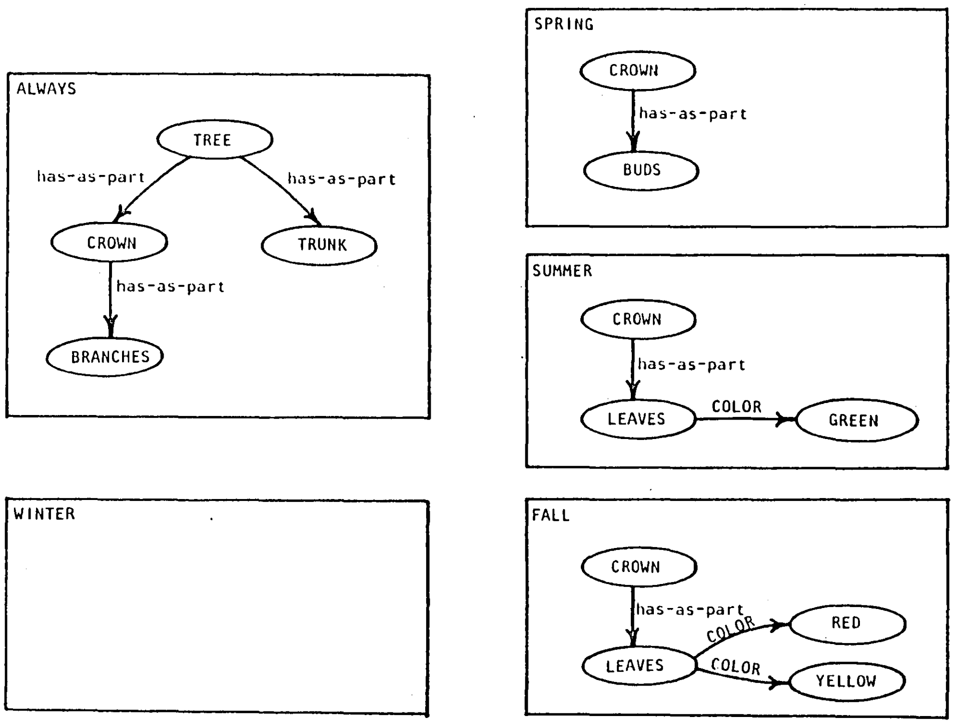
value ϵ {S-expressions representing values}

Note that for any particular portion of a GRAPH there is an infinite number of corresponding GRAPH-DESCRIPTORs and NODE-DESCRIPTORs. This results since GRAPH entities can be included in the descriptors an arbitrary number of times. For edges this includes being described as outpointing and inpointing. If a single entity is described more than once with conflicting values, the final value is the one utilized.

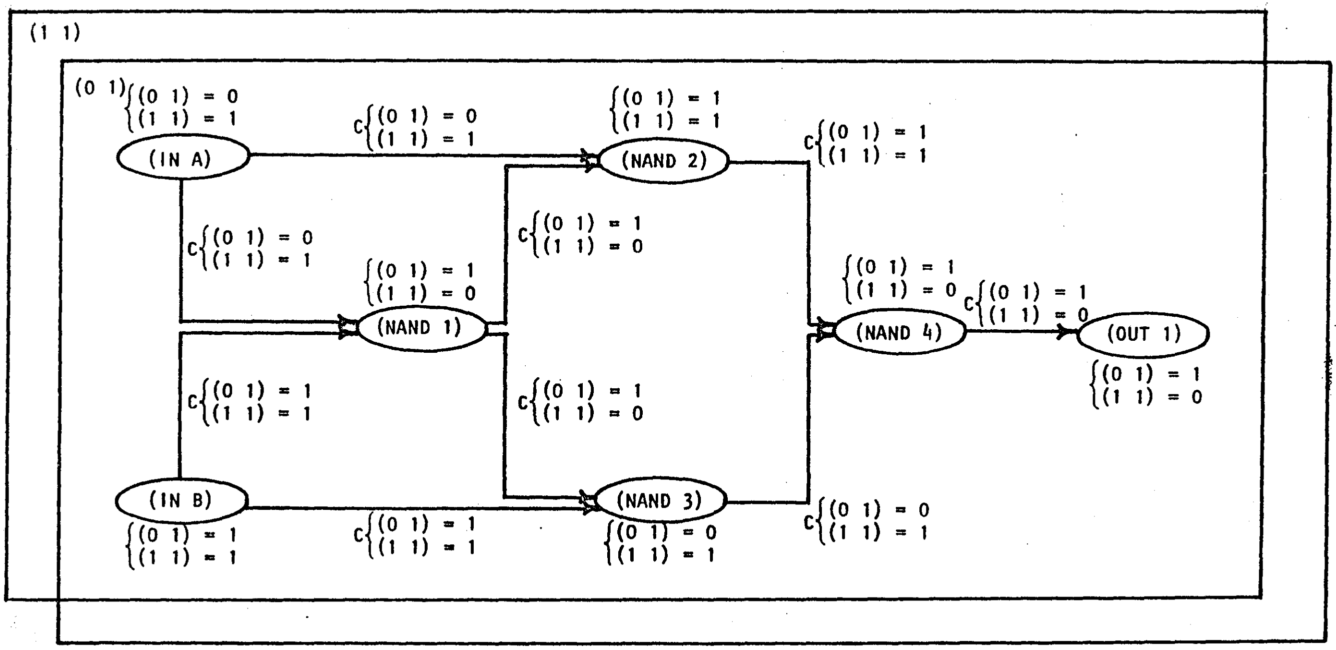
GRAPH-DESCRIPTOR Illustrations



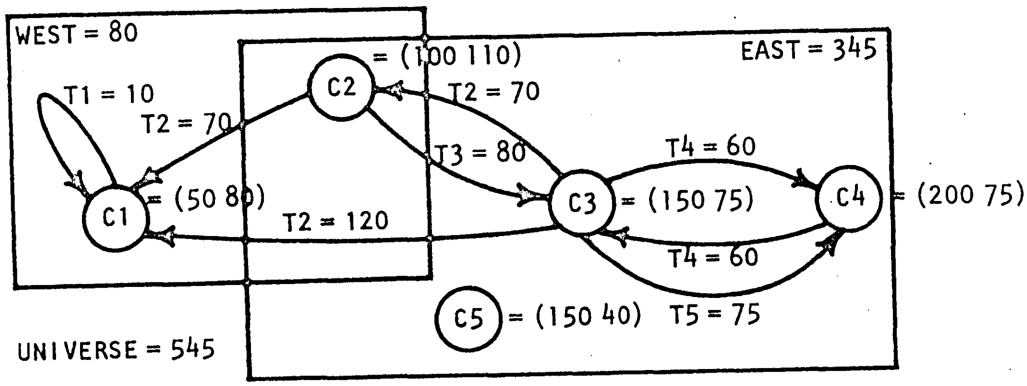

```
(NIL
 (S NIL ((1 NP1) (2 VP)))
 (NP1 NIL ((1 ART) (2 N)))
 (ART NIL ((S THE)))
 (N NIL ((S UNICORN)))
 (VP NIL ((1 V) (2 NP2)))
 (V NIL ((1 V-STEM) (2 PAST)))
 (V-STEM NIL ((S KISS)))
 (PAST NIL ((S ED)))
 (NP2 NIL ((1 PRO)))
 (PRO NIL ((S YOU)))
 (THE)
 (UNICORN)
 (KISS)
 (ED)
 (YOU))
```



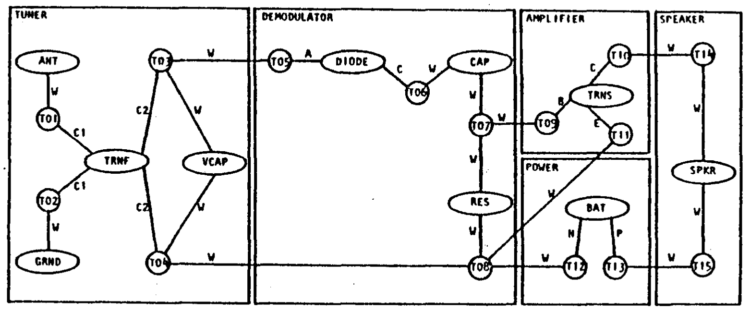

```
((ALWAYS FALL SPRING SUMMER WINTER)
 (BRANCHES (ALWAYS))
 (BUDS (SPRING))
 (CROWN
  (ALWAYS FALL SPRING SUMMER)
  ((HAS-AS-PART BRANCHES (ALWAYS))
   (HAS-AS-PART BUDS (SPRING))
   (HAS-AS-PART LEAVES (FALL SUMMER))))
 (GREEN (SUMMER))
 (LEAVES
  (FALL SUMMER)
  ((COLOR GREEN (SUMMER))
   (COLOR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




```
(((0 1) (1 1))
 ((IN A)
  ((0 1) = 0 (1 1) = 1)
  ((C (NAND 1) ((0 1) = 0 (1 1) = 1))
   (C (NAND 2) ((0 1) = 0 (1 1) = 1))))
 ((IN B)
  ((0 1) = 1 (1 1) = 1)
  ((C (NAND 1) ((0 1) = 1 (1 1) = 1))
   (C (NAND 2) ((0 1) = 1 (1 1) = 1))))
 ((NAND 1)
  ((0 1) = 1 (1 1) = 0)
  ((C (NAND 2) ((0 1) = 1 (1 1) = 0))
   (C (NAND 3) ((0 1) = 1 (1 1) = 0))))
 ((NAND 2) ((0 1) = 1 (1 1) = 1)
           ((C (NAND 4) ((0 1) = 1 (1 1) = 1))))
 ((NAND 3) ((0 1) = 0 (1 1) = 1)
           ((C (NAND 4) ((0 1) = 0 (1 1) = 1))))
 ((NAND 4) ((0 1) = 1 (1 1) = 0)
           ((C (OUT 1) ((0 1) = 1 (1 1) = 0))))
 ((OUT 1) ((0 1) = 1 (1 1) = 0)))
```




```
((EAST = 345 WEST = 80 UNIVERSE = 545)
 (C1 (WEST UNIVERSE = (50 80))
     ((T1 C1 (WEST UNIVERSE = 10))))
 (C2
  (EAST WEST UNIVERSE = (100 110))
  ((T2 C1 (WEST UNIVERSE = 70))
   (T3 C3 (EAST UNIVERSE = 80))))
 (C3
  (EAST UNIVERSE = (150 75))
  ((T2 C1 (UNIVERSE = 120))
   (T2 C2 (EAST UNIVERSE = 70))
   (T4 C4 (EAST UNIVERSE = 60))
   (T5 C4 (EAST UNIVERSE = 75))))
 (C4 (EAST UNIVERSE = (200 75))
     ((T4 C3 (EAST UNIVERSE = 60))))
 (C5 (EAST UNIVERSE = (150 40))))
```




```
((AMPLIFIER DEMODULATOR POWER SPEAKER TUNER)
 (ANT (TUNER)
      ((W T01 (TUNER)))
      ((W T01 (TUNER))))
 (BAT (POWER)
      ((N T12 (POWER)) (P T13 (POWER)))
      ((N T12 (POWER)) (P T13 (POWER))))
 (CAP
  (DEMODULATOR)
  ((W T06 (DEMODULATOR)) (W T07 (DEMODULATOR)))
  ((W T06 (DEMODULATOR)) (W T07 (DEMODULATOR))))
 (DIODE
  (DEMODULATOR)
  ((A T05 (DEMODULATOR)) (C T06 (DEMODULATOR)))
  ((A T05 (DEMODULATOR)) (C T06 (DEMODULATOR))))
 (GRND (TUNER)
       ((W T02 (TUNER)))
       ((W T02 (TUNER))))
 (RES
  (DEMODULATOR)
  ((W T07 (DEMODULATOR)) (W T08 (DEMODULATOR)))
  ((W T07 (DEMODULATOR)) (W T08 (DEMODULATOR))))
 (SPKR (SPEAKER)
       ((W T14 (SPEAKER)) (W T15 (SPEAKER)))
       ((W T14 (SPEAKER)) (W T15 (SPEAKER))))
 (T01 (TUNER)
      ((C1 TRNF (TUNER)) (W ANT (TUNER)))
      ((C1 TRNF (TUNER)) (W ANT (TUNER))))
 (T02 (TUNER)
      ((C1 TRNF (TUNER)) (W GRND (TUNER)))
      ((C1 TRNF (TUNER)) (W GRND (TUNER))))
 (T03
  (TUNER)
  ((C2 TRNF (TUNER)) (W T05) (W VCAP (TUNER)))
  ((C2 TRNF (TUNER)) (W T05) (W VCAP (TUNER))))
 (T04
  (TUNER)
  ((C2 TRNF (TUNER)) (W T08) (W VCAP (TUNER)))
  ((C2 TRNF (TUNER)) (W T08) (W VCAP (TUNER))))
 (T05 (DEMODULATOR)
      ((A DIODE (DEMODULATOR)) (W T03))
      ((A DIODE (DEMODULATOR)) (W T03)))
 (T06 (DEMODULATOR)
      ((C DIODE (DEMODULATOR))
       (W CAP (DEMODULATOR)))
      ((C DIODE (DEMODULATOR))
       (W CAP (DEMODULATOR))))
```

-- continued on next page --

```
(T07
 (DEMODULATOR)
 ((W CAP (DEMODULATOR))
  (W RES (DEMODULATOR))
  (W T09))
 ((W CAP (DEMODULATOR))
  (W RES (DEMODULATOR))
  (W T09)))
(T08
 (DEMODULATOR)
 ((W RES (DEMODULATOR)) (W T04) (W T11))
 ((W RES (DEMODULATOR)) (W T04) (W T11)))
(T09 (AMPLIFIER)
     ((B TRNS (AMPLIFIER)) (W T07))
     ((B TRNS (AMPLIFIER)) (W T07)))
(T10 (AMPLIFIER)
     ((C TRNS (AMPLIFIER)) (W T14))
     ((C TRNS (AMPLIFIER)) (W T14)))
(T11 (AMPLIFIER)
     ((E TRNS (AMPLIFIER)) (W T08))
     ((E TRNS (AMPLIFIER)) (W T08)))
(T12 (POWER)
     ((P BAT (POWER)))
     ((P BAT (POWER))))
(T13 (POWER)
     ((N BAT (POWER)) (W T15))
     ((N BAT (POWER)) (W T15)))
(T14 (SPEAKER)
     ((W SPKR (SPEAKER)) (W T10))
     ((W SPKR (SPEAKER)) (W T10)))
(T15 (SPEAKER)
     ((W SPKR (SPEAKER)) (W T13))
     ((W SPKR (SPEAKER)) (W T13)))
(TRNF
 (TUNER)
 ((C1 T01 (TUNER))
  (C1 T02 (TUNER))
  (C2 T03 (TUNER))
  (C2 T04 (TUNER)))
 ((C1 T01 (TUNER))
  (C1 T02 (TUNER))
  (C2 T03 (TUNER))
  (C2 T04 (TUNER))))
(TRNS
 (AMPLIFIER)
 ((B T09 (AMPLIFIER))
  (C T10 (AMPLIFIER))
  (E T11 (AMPLIFIER)))
 ((B T09 (AMPLIFIER))
  (C T10 (AMPLIFIER))
  (E T11 (AMPLIFIER))))
(VCAP (TUNER)
      ((W T03 (TUNER)) (W T04 (TUNER)))
      ((W T03 (TUNER)) (W T04 (TUNER))))
```

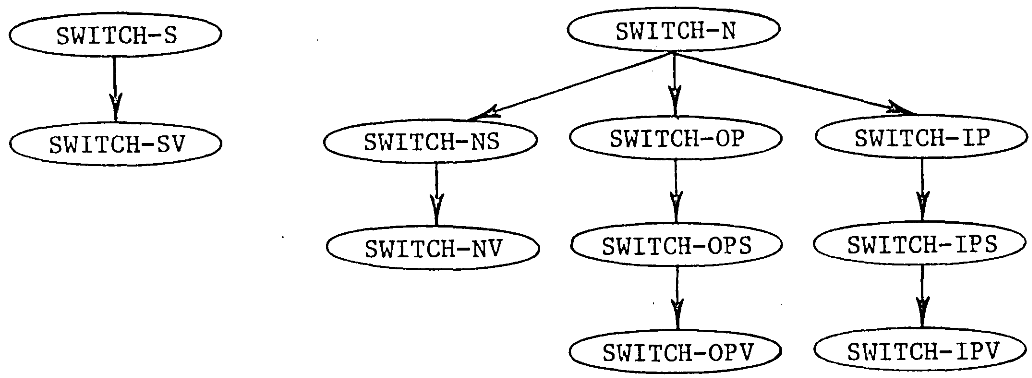
Group II Switches

Many of the Group II operators are sensitive to a collection of global variables refered to as the Group II switches. Each switch corresponds to a graph entity type. The switches and their corresponding types are shown here.

SWITCH-S - spaces
SWITCH-N - nodes
SWITCH-OP - outpointing pairs
SWITCH-IP - inpointing pairs
SWITCH-SV - space values
SWITCH-NV - node values
SWITCH-OPV - outpointing pair values
SWITCH-IPV - inpointing pair values
SWITCH-NS - spaces of a node
SWITCH-OPS - spaces of an outpointing pair
SWITCH-IPS - spaces of an inpointing pair

Initially all switches are "on" (i.e., have a non-NIL LISP value). When a switch is "off" (i.e., has a LISP value of NIL), certain aspects of Group II operators, corresponding to the switches' associated type, are inhibited. For example, when the switch controlling nodes (SWITCH-N) is off, nodes are not created by CREATE operators, described by DESCRIBE operators, or printed by PRINT operators.

Some switches take precedence over other switches resulting in the following hierarchy.



When a switch is off, all those switches below it in the hierarchy are implicitly off; that is, all Group II operators will function as if they are also off regardless of their own setting. For example, when the space switch (SWITCH-S) is off, neither spaces nor space values (SWITCH-SV) will be described by DESCRIBE operators.

Rules of Group II Operator Composition

1. The name of each Group II operator is a hyphenated compound word.

   e.g., CREATE-GRAPH
   PRINT-SPACE

2. The role played by each word element is determined by its position in the operator's name as follows.

   <operator type> - <operator object>

3. There are two major categories of operators, functions and pseudo-functions. Functions are executed for the value they return. Pseudo-functions are executed for their effect. Group II pseudo-functions all return T. These categories have the following associated operator types.

pseudo-functions	CREATE	= creates the specified portion of a GRAPH if it does not already exist
	DESTROY	= destroys the specified portion of the GRAPH if it exists
	PRINT	= prints the specified portion of the GRAPH
functions	DESCRIBE	= returns a descriptor of the specified portion of the GRAPH

4. The operator object specifies the type of GRAPH entity the operator manipulates. These include the following.

   GRAPH - e.g., DESTROY-GRAPH destroys the current GRAPH

   SPACE - e.g., PRINT-SPACE prints a description of a space in the current GRAPH

   NODE - e.g., DESCRIBE-NODE returns a node-descriptor for a node in the current GRAPH

5. All combinations of operator types and operator objects are included in the Group II operators except the following.

   "CREATE-SPACE" - CREATE-GRAPH accomplishes the same

   "DESTROY-SPACE" - the Group I operator DUS accomplishes the same

   "DESTROY-NODE" - the Group I operator DUN accomplishes the same

On the following two pages are two views of all Group II operators.

Group II GRASPER Operators: Tabular Summary

(CREATE-GRAPH *graph-descriptor*)

(CREATE-NODE *node-descriptor*)

(DESCRIBE-GRAPH)

(DESCRIBE-SPACE *space*)

(DESCRIBE-NODE *node*)(DESCRIBE-NODE *node list-of-spaces*)

(DESTROY-GRAPH)

(PRINT-GRAPH)

(PRINT-SPACE *space*)

(PRINT-NODE *node*)(PRINT-NODE *node list-of-spaces*)

Operator Composition

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

Group II Operator Descriptions

This section contains a complete description of each Group II operator in alphabetical order. Each description consists of

1) the calling form (in LISP syntax) of the operator with its arguments,
2) its informal definition including
   (a) a prose description of the operator's purpose,
   and (b) a prose description of each GRASPER error condition,
3) its formal definition including all GRASPER error conditions,
and 4) a group of illustrations (in LISP syntax) including the generation of each GRASPER error condition.

Each group of illustrations begins with a drawing of the GRAPH which exists before each illustrative call. The series of calls does not represent a sequence during one user session. If a call alters the GRAPH, a drawing of the resulting GRAPH is given.

Some of the following formal definitions include the concatenation operator, ⊕. Ordered sequences of entities are represented by enclosing them in angle-brackets, < >. Concatenating such a sequence to another entity has the effect of removing the angle-brackets.

e.g., ONE ⊕ TWO ⊕ THREE = ONE TWO THREE
<ONE TWO> ⊕ <THREE> ⊕ < > = ONE TWO THREE

Some of the formal definitions include PRINT and SET-LEFT-MARGIN as operators. PRINT takes any number of arguments and outputs them to the current device followed by a carriage return and line feed (additional carriage returns and line feeds occur during a PRINT whenever the print line is full). SET-LEFT-MARGIN has one integer argument which indicates the position on the line the carriage should assume after a carriage return. If the carriage position is to the left of the position indicated when a SET-LEFT-MARGIN occurs, the carriage is advanced to that position.

```
e.g., PRINT[THIS,IS]
      SET-LEFT-MARGIN[2]          THIS IS
      PRINT[AN,EXAMPLE]     --->    AN EXAMPLE
      SET-LEFT-MARGIN[0]
      PRINT[ ]                    OF THE OUTPUT
      PRINT[OF,THE,OUTPUT]
```

The descriptions follow in alphabetical order.

(CREATE-GRAPH *graph-descriptor*)

Informal Definition

The pseudo-function CREATE-GRAPH is an EXPR which has the effect of adding the GRAPH described by *graph-descriptor* to the existing GRAPH. Values of GRAPH entities described in *graph-descriptor* override conflicting values in the existing GRAPH. CREATE-GRAPH does not create those entities whose corresponding switches are off. CREATE-GRAPH has no effect if all the entities and values described in *graph-descriptor* already exist. CREATE-GRAPH returns T.

error conditions:

- *graph-descriptor* does not structurally conform to the GRAPH-DESCRIPTOR definition
- *graph-descriptor* references a nonexisting node or space not described in *graph-descriptor*

Formal Definition

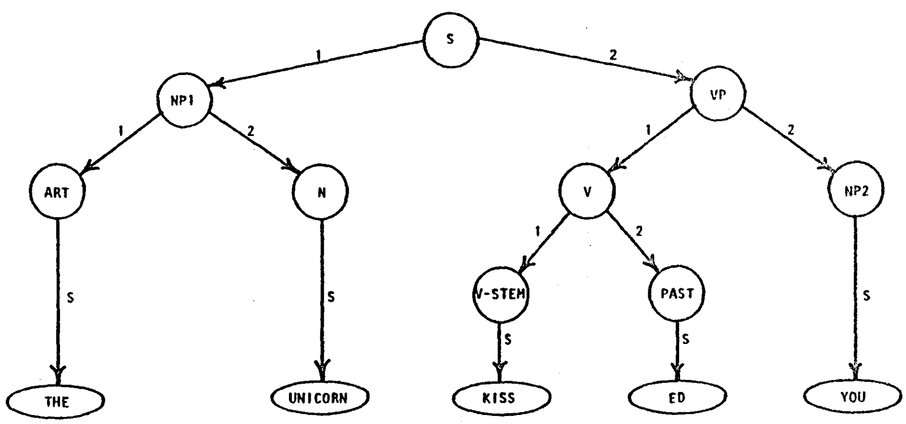
CREATE-GRAPH[grd] = T
with effects:
where grd = (svdl nd_1 ... nd_i ... nd_t)
if SWITCH-S ≠ NIL
then create-s[svdl]
if SWITCH-N ≠ NIL
then for each nd_i CREATE-NODE[nd_i]

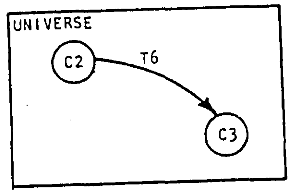
-- continued on next page --

```
create-s[svdl]
   with effects:
      where svdl = (svd_1 ... svd_i ... svd_t)
         for each svd_i = <s_i>|<s_i = v_i>
            CUS[s_i]
            if SWITCH-SV ≠ NIL and ∃v_i
               then BUS[s_i, v_i]
```

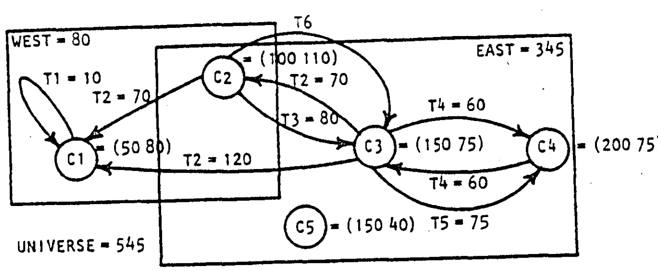
Illustrations

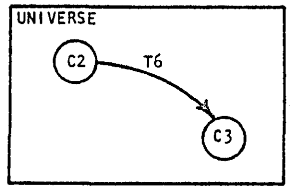

```
?(PROGN
  (PRINT-SWITCHES)
  (CREATE-GRAPH
   '(NIL
    (S NIL ((1 NP1) (2 VP)))
    (NP1 NIL ((1 ART) (2 N)))
    (ART NIL ((S THE)))
    (N NIL ((S UNICORN)))
    (VP NIL ((1 V) (2 NP2)))
    (V NIL ((1 V-STEM) (2 PAST)))
    (V-STEM NIL ((S KISS)))
    (PAST NIL ((S ED)))
    (NP2 NIL ((1 PRO)))
    (PRO NIL ((S YOU)))
    (THE)
    (UNICORN)
    (KISS)
    (ED)
    (YOU))))

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

T
```



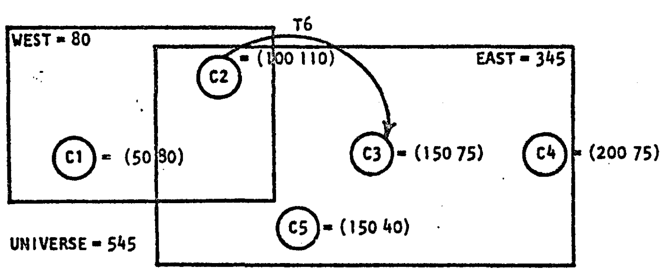


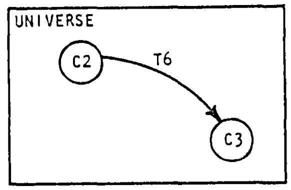

```
?(PROGN
  (PRINT-SWITCHES)
  (CREATE-GRAPH
   '((EAST = 345 WEST = 80 UNIVERSE = 545)
     (C1 (WEST UNIVERSE = (50 80)) ((T1 C1 (WEST UNIVERSE = 10))))
     (C2 (EAST WEST UNIVERSE = (100 110))
         ((T2 C1 (WEST UNIVERSE = 70)) (T3 C3 (EAST UNIVERSE = 80))))
     (C3
      (EAST UNIVERSE = (150 75))
      ((T2 C1 (UNIVERSE = 120))
       (T2 C2 (EAST UNIVERSE = 70))
       (T4 C4 (EAST UNIVERSE = 60))
       (T5 C4 (EAST UNIVERSE = 75))))
     (C4 (EAST UNIVERSE = (200 75)) ((T4 C3 (EAST UNIVERSE = 60))))
     (C5 (EAST UNIVERSE = (150 40))))))

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

T
```






```
?(PROGN
  (PRINT-SWITCHES)
  (CREATE-GRAPH
   '((EAST = 345 WEST = 80 UNIVERSE = 545)
    (C1 (WEST UNIVERSE = (50 80)) ((T1 C1 (WEST UNIVERSE = 10))))
    (C2 (EAST WEST UNIVERSE = (100 110))
        ((T2 C1 (WEST UNIVERSE = 70)) (T3 C3 (EAST UNIVERSE = 80))))
    (C3
     (EAST UNIVERSE = (150 75))
     ((T2 C1 (UNIVERSE = 120))
      (T2 C2 (EAST UNIVERSE = 70))
      (T4 C4 (EAST UNIVERSE = 60))
      (T5 C4 (EAST UNIVERSE = 75))))
    (C4 (EAST UNIVERSE = (200 75)) ((T4 C3 (EAST UNIVERSE = 60))))
    (C5 (EAST UNIVERSE = (150 40))))))

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

T
```






```
?(CREATE-GRAPH 'GDX)

*** CREATE-GRAPH ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                        THE GRAPH-DESCRIPTOR WAS GDX

?(CREATE-GRAPH '(SVDX))

*** CREATE-GRAPH ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                        BAD SPACE-VALUE-DESCRIPTOR
                        THE SPACE-VALUE-DESCRIPTOR WAS SVDX

?(CREATE-GRAPH '(NIL NDX))

*** CREATE-GRAPH ERROR: POORLY FORMED NODE-DESCRIPTOR
                        THE NODE-DESCRIPTOR WAS NDX

?(CREATE-GRAPH '(NIL (N1 NSVDLX)))

*** CREATE-GRAPH ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                        BAD NODE-SPACE-VALUE-DESCRIPTOR ASSOCIATED
                        WITH NODE N1
                        THE NODE-SPACE-VALUE-DESCRIPTOR WAS NSVDLX

?(CREATE-GRAPH '(NIL (N1 (SX))))

*** CREATE-GRAPH ERROR: SX IS NOT A SPACE

?(CREATE-GRAPH '(NIL (N1 NIL OPDLX)))

*** CREATE-GRAPH ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                        BAD LIST OF OUTPOINTING-PAIR-DESCRIPTOR
                        ASSOCIATED WITH NODE N1
                        THE OUTPOINTING-PAIR-DESCRIPTOR LIST WAS OPDLX

?(CREATE-GRAPH '(NIL (N1 NIL ((G NX)))))

*** CREATE-GRAPH ERROR: NX IS NOT A NODE

?(CREATE-GRAPH '(NIL (N1 NIL ((G N1 PSVDLX)))))

*** CREATE-GRAPH ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                        BAD OUTPOINTING-PAIR-DESCRIPTOR ASSOCIATED
                        WITH NODE N1
                        THE OUTPOINTING-PAIR-DESCRIPTOR WAS (G N1 PSVDLX)

?(CREATE-GRAPH '(NIL (N1 ((G N1 (SX))))))

*** CREATE-GRAPH ERROR: SX IS NOT A SPACE
```

(CREATE-NODE *node-descriptor*)

Informal Definition

The pseudo-function CREATE-NODE is an EXPR which has the effect of adding a partial GRAPH described by *node-descriptor* to the existing GRAPH. Values of GRAPH entities described in *node-descriptor* override conflicting values in the existing GRAPH. CREATE-NODE does not create those entities whose corresponding switches are off. CREATE-NODE has no effect if all the entities and values described in *node-descriptor* already exist. CREATE-NODE returns T.

error conditions:
- *node-descriptor* does not structurally conform to the NODE-DESCRIPTOR definition
- *node-descriptor* references a nonexisting space or node (other than the primary node of *node-descriptor*)

Formal Definition

```
CREATE-NODE[nd] = T
   with effects:
      where nd = (n)|
                 (n  nsvdl)|
                 (n  nsvdl  opdl)|
                 (n  nsvdl  opdl  ipdl)
         CUN[n]
         if SWITCH-NS ≠ NIL and ∃nsvdl
            then create-nsvdl[n, nsvdl]
         if SWITCH-OP ≠ NIL and ∃opdl
            then create-opdl[n, opdl]
         if SWITCH-IP ≠ NIL and ∃ipdl
            then create-ipdl[n, ipdl]
```


-- continued on next page --

---

create-nsvdl[n, nsvdl]

  with effects:

    where nsvdl = $(nsvd_1 \oplus \ldots \oplus nsvd_i \oplus \ldots \oplus nsvd_t)$

      for each $nsvd_i = \langle s_i \rangle \mid \langle s_i \ = \ v_i \rangle$

        CUN[n, s_i]

        if SWITCH-NV $\neq$ NIL and $\exists v_i$

          then BUN[n, v_i, s_i]

---

create-opdl[n, opdl]

  with effects:

    where opdl = $(opd_1 \ \ldots \ opd_i \ \ldots \ opd_t)$

      for each $opd_i = (g_i \ \ m_i) \mid$

        $(g_i \ \ m_i \ \ opsvdl_i)$

        COP[n, g_i, m_i]

        if SWITCH-OPS $\neq$ NIL and $\exists opsvdl_i$

          then create-opsvdl[n, g_i, m_i, $opsvdl_i$]

---

create-opsvdl[n, g, m, opsvdl]

  with effects:

    where opsvdl = $(opsvd_1 \oplus \ldots \oplus opsvd_i \oplus \ldots \oplus opsvd_t)$

      for each $opsvd_i = \langle s_i \rangle \mid \langle s_i \ = \ v_i \rangle$

        COP[n, g, m, s_i]

        if SWITCH-OPV $\neq$ NIL and $\exists v_i$

          then BOP[n, g, m, v_i, s_i]

---

create-ipdl[n, ipdl]

  with effects:

    where ipdl = $(ipd_1 \ \ldots \ ipd_i \ \ldots \ ipd_t)$

      for each $ipd_i = (g_i \ \ m_i) \mid$

        $(g_i \ \ m_i \ \ ipsvdl_i)$

        CIP[n, g_i, m_i]

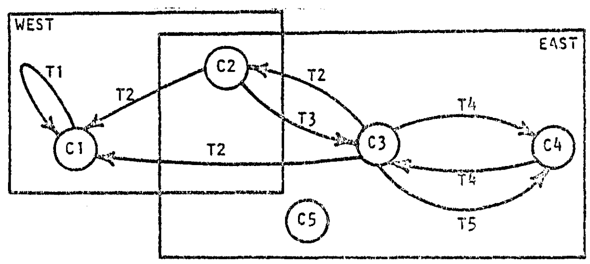
        if SWITCH-IPS $\neq$ NIL and $\exists ipsvdl_i$

          then create-ipsvdl[n, g_i, m_i, $ipsvdl_i$]

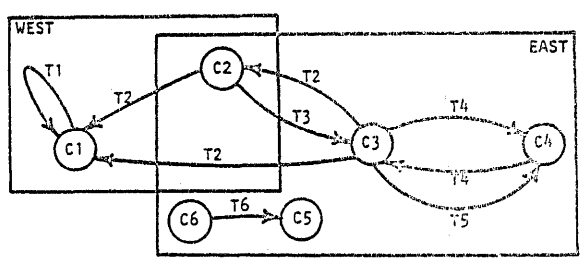
-- continued on next page --

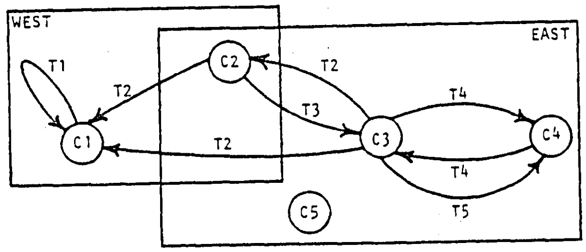
create-ipsvdl[n, g, m, ipsvdl]
with effects:
where ipsvdl = (ipsvd$_1$ ⊕ ... ⊕ ipsvd$_i$ ⊕ ... ⊕ ipsvd$_t$)
for each ipsvd$_i$ = <s$_i$>|<s$_i$ = v$_i$>
CIP[n, g, m, s$_i$]
if SWITCH-IPV ≠ NIL and ∃v$_i$
then BIP[n, g, m, v$_i$, s$_i$]

Illustrations

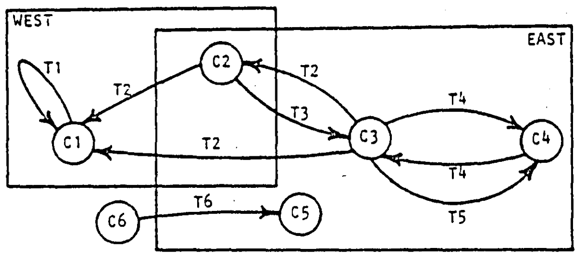



```
?(PROGN (PRINT-SWITCHES) (CREATE-NODE '(C6 (EAST) ((T6 C5 (EAST))))))

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

NIL
```






```
?(PROGN (PRINT-SWITCHES) (CREATE-NODE '(C6 (EAST) ((T6 C5 (EAST))))))

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 = T                  SWITCH-IPV = T
```




```
?(CREATE-NODE '(N1 NSVDLX))
*** CREATE-NODE ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                       BAD NODE-SPACE-VALUE-DESCRIPTOR ASSOCIATED WITH
                       NODE N1
                       THE NODE-SPACE-VALUE-DESCRIPTOR WAS NSVDLX

?(CREATE-NODE 'NDX)
*** CREATE-NODE ERROR: POORLY FORMED NODE-DESCRIPTOR
                       THE NODE-DESCRIPTOR WAS NDX

?(CREATE-NODE '(N1 NSVDLX))
*** CREATE-NODE ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                       BAD NODE-SPACE-VALUE-DESCRIPTOR ASSOCIATED WITH
                       NODE N1
                       THE NODE-SPACE-VALUE-DESCRIPTOR WAS NSVDLX

?(CREATE-NODE '(N1 (SX)))
*** CREATE-NODE ERROR: SX IS NOT A SPACE

?(CREATE-NODE '(N1 NIL OPDLX))
*** CREATE-NODE ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                       BAD LIST OF OUTPOINTING-PAIR-DESCRIPTORS
                       ASSOCIATED WITH NODE N1
                       THE OUTPOINTING-PAIR-DESCRIPTOR LIST WAS OPDLX

?(CREATE-NODE '(N1 NIL ((G NX))))
*** CREATE-NODE ERROR: NX IS NOT A NODE

?(CREATE-NODE '(N1 NIL ((G N1 PSVDLX))))
*** CREATE-NODE ERROR: POORLY FORMED GRAPH-DESCRIPTOR
                       BAD OUTPOINTING-PAIR-DESCRIPTOR ASSOCIATED WITH
                       NODE N1
                       THE OUTPOINTING-PAIR-DESCRIPTOR WAS (G N1 PSVDLX)

?(CREATE-NODE '(N1 NIL ((G N1 (SX)))))
*** CREATE-NODE ERROR: SX IS NOT A SPACE
```

(DESCRIBE-GRAPH)

Informal Definition

The function DESCRIBE-GRAPH is an EXPR which returns a GRAPH-DESCRIPTOR for the existing GRAPH. DESCRIBE-GRAPH does not describe NIL values, inclusion in the universal space when the universal value is NIL, or entities whose corresponding switches are off.

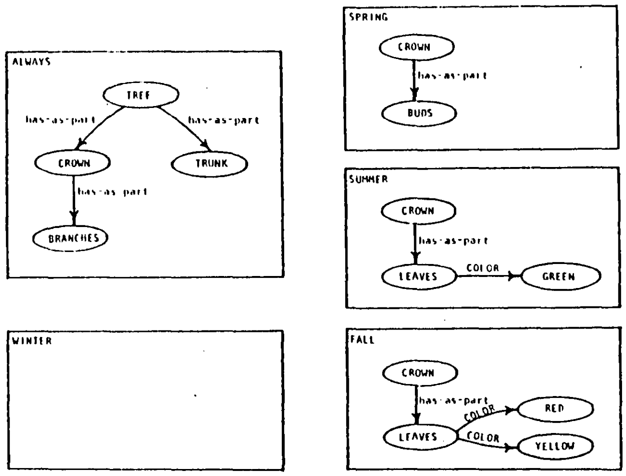
Formal Definition

```
DESCRIBE-GRAPH[ ] = (sdl ⊕ nds)
   where
      if SWITCH-S ≠ NIL
         then sdl = describe-ss[ ]
         else sdl = NIL
      if SWITCH-N ≠ NIL
         then nds = describe-ns[ ]
         else nds = < >
```

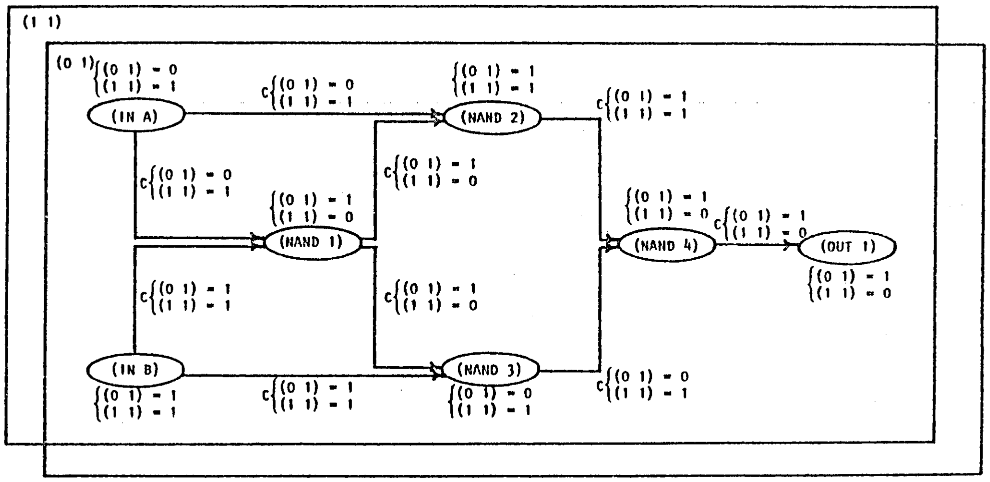
```
describe-ss[ ] = (svd_1 ⊕ ... ⊕ svd_i ⊕ ... ⊕ svd_t)
   where
      for each s_i ∈ SUS[ ] ∪ {UNIVERSE}
         if SWITCH-SV ≠ NIL and VUS[s_i] ≠ NIL
            then svd_i = <s_i = VUS[s_i]>
            else if s_i ≠ UNIVERSE
                    then svd_i = <s_i>
                    else svd_i = < >
```

```
describe-ns[ ] = <nd_1 ... nd_i ... nd_t>
   where
      for each n_i ∈ SUN[ ]
         nd_i = DESCRIBE-NODE[n_i]
```

Illustrations

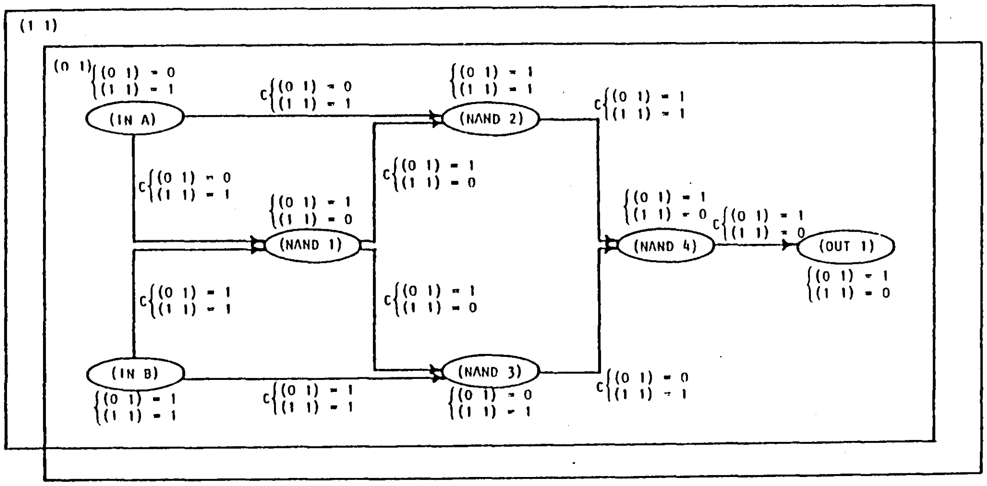



```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-GRAPH))

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

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




```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-GRAPH))

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

(((0 1) (1 1))
 ((IN A) ((0 1) = 0 (1 1) = 1)
         ((C (NAND 1) ((0 1) = 0 (1 1) = 1))
          (C (NAND 2) ((0 1) = 0 (1 1) = 1))))
 ((IN B) ((0 1) = 1 (1 1) = 1)
         ((C (NAND 1) ((0 1) = 1 (1 1) = 1))
          (C (NAND 2) ((0 1) = 1 (1 1) = 1))))
 ((NAND 1)
  ((0 1) = 1 (1 1) = 0)
  ((C (NAND 2) ((0 1) = 1 (1 1) = 0))
   (C (NAND 3) ((0 1) = 1 (1 1) = 0)))
  ((C (IN A) ((0 1) = 0 (1 1) = 1)) (C (IN B) ((0 1) = 1 (1 1) = 1))))
 ((NAND 2)
  ((0 1) = 1 (1 1) = 1)
  ((C (NAND 4) ((0 1) = 1 (1 1) = 1)))
  ((C (IN A) ((0 1) = 0 (1 1) = 1))
   (C (IN B) ((0 1) = 1 (1 1) = 1))
   (C (NAND 1) ((0 1) = 1 (1 1) = 0))))
 ((NAND 3) ((0 1) = 0 (1 1) = 1)
           ((C (NAND 4) ((0 1) = 0 (1 1) = 1)))
           ((C (NAND 1) ((0 1) = 1 (1 1) = 0))))
 ((NAND 4)
  ((0 1) = 1 (1 1) = 0)
  ((C (OUT 1) ((0 1) = 1 (1 1) = 0)))
  ((C (NAND 2) ((0 1) = 1 (1 1) = 1))
   (C (NAND 3) ((0 1) = 0 (1 1) = 1))))
 ((OUT 1) ((0 1) = 1 (1 1) = 0)
          NIL
          ((C (NAND 4) ((0 1) = 1 (1 1) = 0)))))
```

DESCRIBE-GRAPH




```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-GRAPH))

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

(NIL
 ((IN A) ((0 1) = 0 (1 1) = 1)
         ((C (NAND 1) ((0 1) (1 1))) (C (NAND 2) ((0 1) (1 1)))))
 ((IN B) ((0 1) = 1 (1 1) = 1)
         ((C (NAND 1) ((0 1) (1 1))) (C (NAND 2) ((0 1) (1 1)))))
 ((NAND 1)
  ((0 1) = 1 (1 1) = 0)
  ((C (NAND 2) ((0 1) (1 1))) (C (NAND 3) ((0 1) (1 1))))
  ((C (IN A) ((0 1) (1 1))) (C (IN B) ((0 1) (1 1)))))
 ((NAND 2)
  ((0 1) = 1 (1 1) = 1)
  ((C (NAND 4) ((0 1) (1 1))))
  ((C (IN A) ((0 1) (1 1))) (C (IN B) ((0 1) (1 1)))
                            (C (NAND 1) ((0 1) (1 1)))))
 ((NAND 3) ((0 1) = 0 (1 1) = 1)
           ((C (NAND 4) ((0 1) (1 1))))
           ((C (NAND 1) ((0 1) (1 1)))))
 ((NAND 4) ((0 1) = 1 (1 1) = 0)
           ((C (OUT 1) ((0 1) (1 1))))
           ((C (NAND 2) ((0 1) (1 1))) (C (NAND 3) ((0 1) (1 1)))))
 ((OUT 1) ((0 1) = 1 (1 1) = 0) NIL ((C (NAND 4) ((0 1) (1 1))))))
```

(DESCRIBE-NODE *node*)[1]

Informal Definition

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

Formal Definition

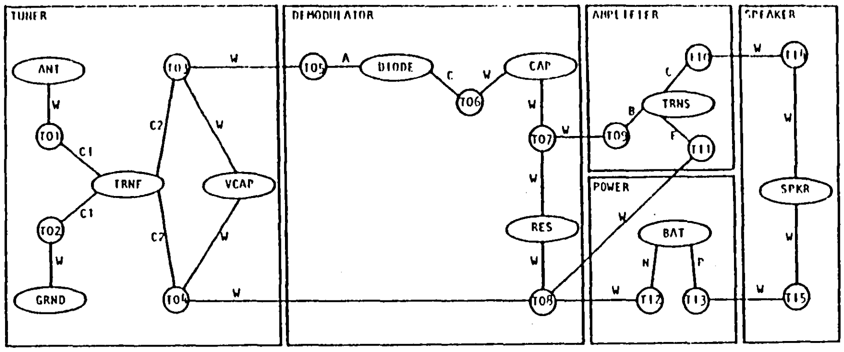
DESCRIBE-NODE[n] = DESCRIBE-NODE[n, sus[n] $\cup$ {UNIVERSE}]

error condition:

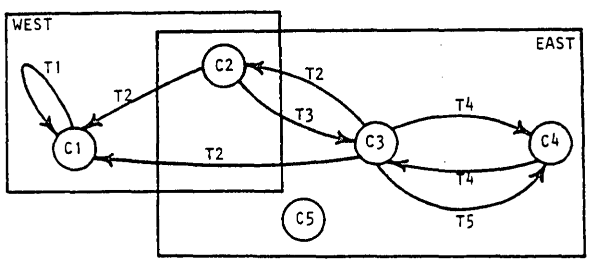
- $n \notin N$

[1]See alternative form on page 284.

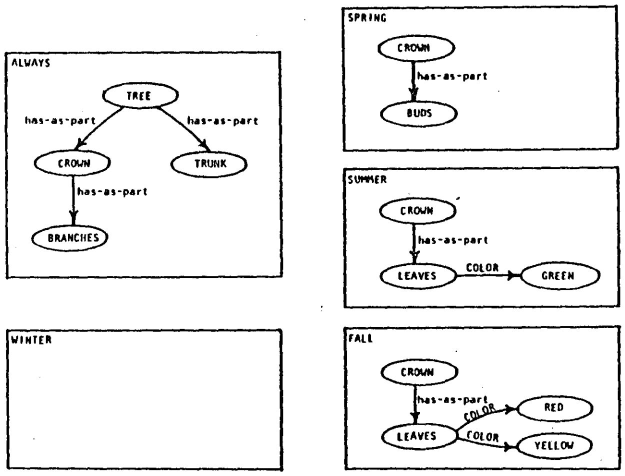
Illustrations



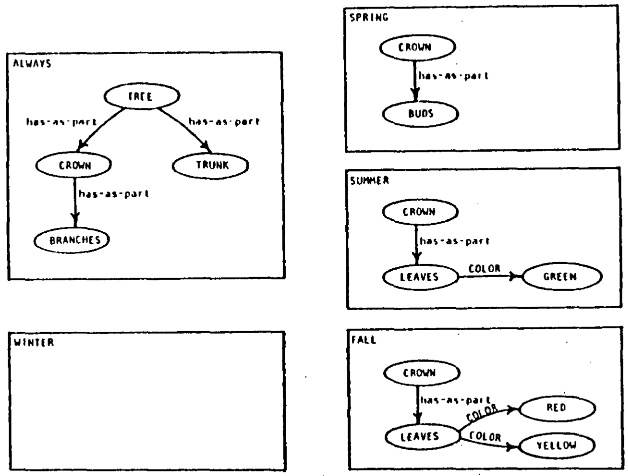

```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'T04))

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))))
```




```
?(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
```

(DESCRIBE-NODE *node list-of-spaces*)[1]

Informal Definition

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

error conditions:

- *node* does not exist in any space in *list-of-spaces*
- some space in *list-of-spaces* does not exist

Formal Definition

DESCRIBE-NODE[n, sl] = (n ⊕ nsvdl ⊕ opdl ⊕ ipdl)

    where sl = (s_1 ... s_i ... s_t)
        if SWITCH-IP ≠ NIL and for some s_i SIP[n, s_i] ≠ NIL
            then ipdl = describe-ip[n, sl]
            else ipdl = < >
        if SWITCH-OP ≠ NIL and for some s_i SOP[n, s_i] ≠ NIL
            then opdl = describe-op[n, sl]
            else if ipdl = < >
                then opdl = < >
                else opdl = NIL
        if SWITCH-NS ≠ NIL and for some s_i XUN[n, s_i] ≠ NIL
            then nsvdl = describe-nsv[n, sl]
            else if ipdl = < > and opdl = < >
                then nsvdl = < >
                else nsvdl = NIL

-- continued on next page --

[1] See alternative form on page 280.

```
describe-nsv[n, sl] = (nsvd_1 ⊕ ... ⊕ nsvd_i ⊕ ... ⊕ nsvd_t)
   where sl = (s_1 ... s_i ... s_t)
      for each s_i
         if XUN[n, s_i] = T
            then if SWITCH-NV ≠ NIL and VUN[n, s_i] ≠ NIL
                    then nsvd_i = <s_i  =  VUN[n, s_i]>
                    else if s_i ≠ UNIVERSE
                            then nsvd_i = <s_i>
                            else nsvd_i = < >
            else nsvd_i = < >
```

```
describe-op[n, sl] = (opd_1 ⊕ ... ⊕ opd_i ⊕ ... ⊕ opd_t)
   where sl = (s_1 ... s_t ... s_u)
      for each (g_i  m_i) ∈ SOP[n]
         if for some s_t XOP[n, g_i, m_i, s_t] = T
            then if SWITCH-OPS ≠ NIL
                    then opd_i = (g_i ⊕ m_i ⊕ opsvl_i)
                         where opsvl_i = describe-opsv[n, g_i, m_i, sl]
                    else opd_i = (g_i  m_i)
            else opd_i = < >
```

```
describe-opsv[n, g, m, sl] = (opsvd_1 ⊕ ... ⊕ opsvd_i ⊕ ... ⊕ opsvd_t)
   where sl = (s_1 ... s_i ... s_t)
      for each s_i
         if XOP[n, g, m, s_i] = T
            then if SWITCH-OPV ≠ NIL and VOP[n, g, m, s_i] ≠ NIL
                    then opsvd_i = <s_i  =  VOP[n, g, m, s_i]>
                    else if s_i ≠ UNIVERSE
                            then opsvd_i = <s_i>
                            else opsvd_i = < >
            else opsvd_i = < >
```

-- continued on next page --

---

describe-ip$[n, sl] = (ipd_1 \oplus \ldots \oplus ipd_i \oplus \ldots \oplus ipd_t)$

where $sl = (s_1 \ldots s_t \ldots s_u)$

for each $(g_i \; m_i) \in SIP[n]$

if for some s_t $XIP[n, g_i, m_i, s_t] = T$

then if SWITCH-IPS $\neq$ NIL

then $ipd_i = (g_i \oplus m_i \oplus ipsvl_i)$

where $ipsvl_i$ = describe-ipsv$[n, g_i, m_i, sl]$

else $ipd_i = (g_i \; m_i)$

else $ipd_i = < >$

---

describe-ipsv$[n, g, m, sl] = (ipsvd_1 \oplus \ldots \oplus ipsvd_i \oplus \ldots \oplus ipsvd_t)$

where $sl = (s_1 \ldots s_i \ldots s_t)$

for each s_i

if $XIP[n, g, m, s_i] = T$

then if SWITCH-IPV $\neq$ NIL and $VIP[n, g, m, s_i] \neq$ NIL

then $ipsvd_i = <s_i = VIP[n, g, m, s_i]>$

else if $s_i \neq$ UNIVERSE

then $ipsvd_i = <s_i>$

else $ipsvd_i = < >$

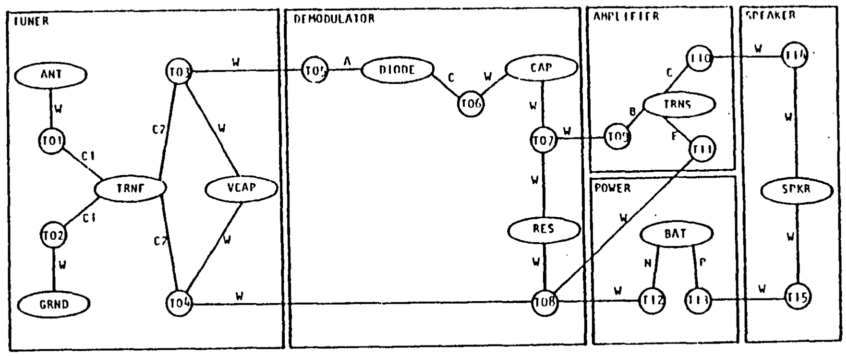
else $ipsvd_i = < >$

---

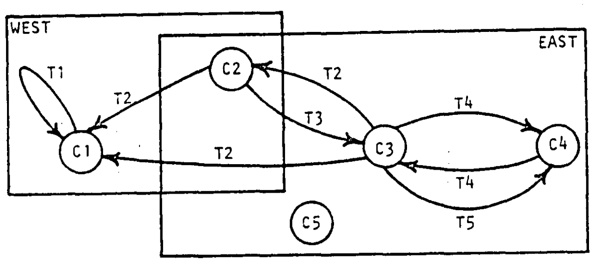
error conditions:

- $((n \; s_i) \; v) \notin NSV$ for all $v \in V$ where $sl = (s_1 \ldots s_i \ldots s_t)$
- some $s_i \notin S$ where $sl = (s_1 \ldots s_i \ldots s_t)$

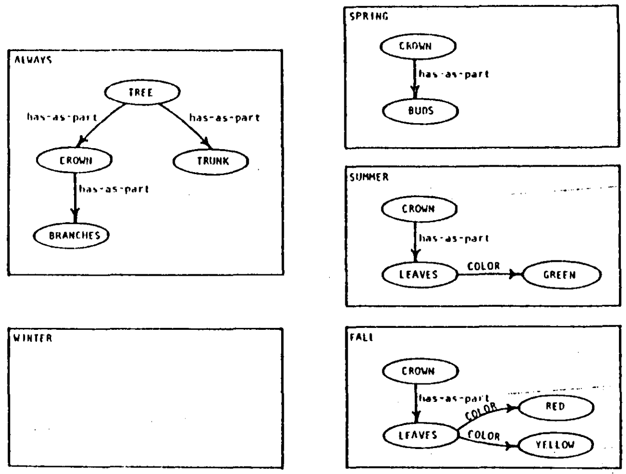
Illustrations




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

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

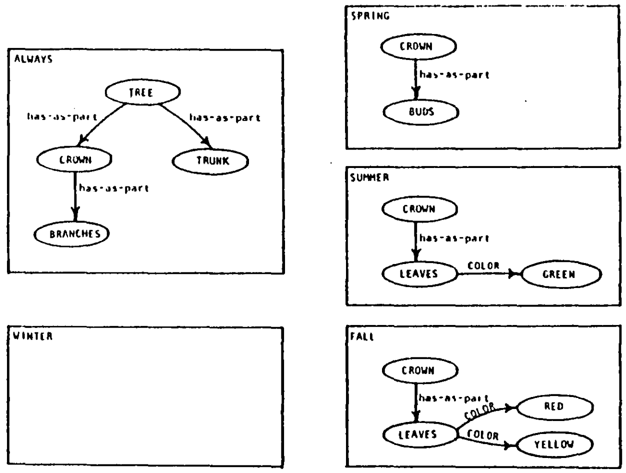



```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-NODE 'C2 '(EAST)))

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) ((T3 C3 (EAST))) ((T2 C3 (EAST))))
```




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

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 SUMMER)
       ((HAS-AS-PART BRANCHES (ALWAYS)) (HAS-AS-PART LEAVES (SUMMER)))
       ((HAS-AS-PART TREE (ALWAYS))))
```

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

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 = NIL         SWITCH-IPS = T             SWITCH-IPV = T

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




```
?(DESCRIBE-NODE 'LEAVES '(WINTER SPRING))
*** DESCRIBE-NODE ERROR: LEAVES IS NOT A NODE IN SPACE (VIRTUAL-SPACE
                                   (WINTER SPRING))[1]
```

```
?(DESCRIBE-NODE 'LEAVES '(SUMMER SX))
*** DESCRIBE-NODE ERROR: SX IS NOT A SPACE
```

[1]See virtual spaces on page 321.

(DESCRIBE-SPACE *space*)

Informal Definition

The function DESCRIBE-SPACE is an EXPR which returns a GRAPH-DESCRIPTOR describing *space* in the existing GRAPH. DESCRIBE-SPACE 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:

- *space* does not exist

Formal Definition

DESCRIBE-SPACE[s] = (sdl ⊕ nds)

where

if SWITCH-S ≠ NIL

then if SWITCH-SV ≠ NIL and VUS[s] ≠ NIL

then sdl = (s = VUS[s])

else sdl = (s)

else sdl = NIL

if SWITCH-N ≠ NIL

then nds = $<nd_1 \ldots nd_i \ldots nd_t>$

where for each $n_i \in$ SUN[s]

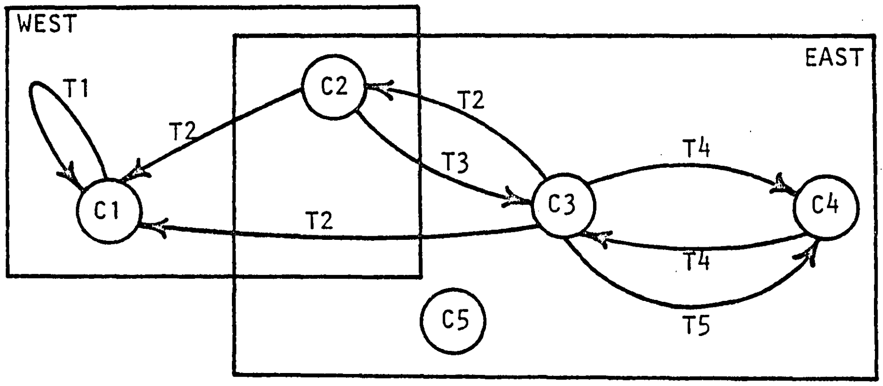
nd_i = DESCRIBE-NODE[n_i, (s)]

else nds = < >

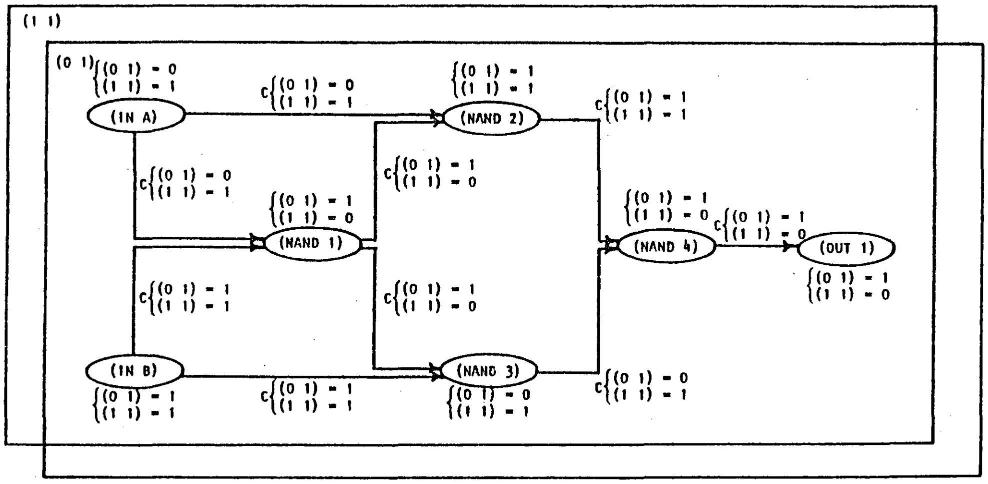
error condition:

- s ∉ S

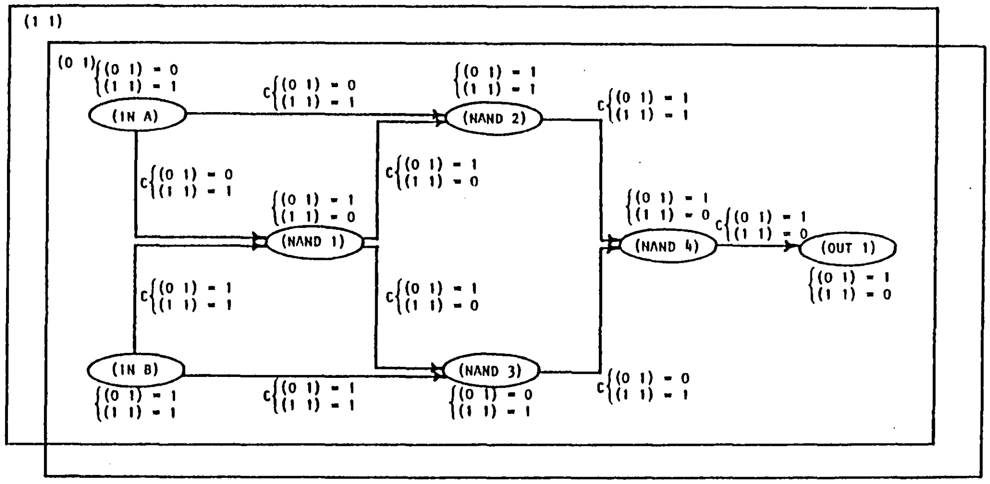
Illustrations




```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-SPACE 'EAST))

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

((EAST = 345)
 (C2 (EAST) ((T3 C3 (EAST))) ((T2 C3 (EAST))))
 (C2 (EAST)
     ((T2 C2 (EAST)) (T4 C4 (EAST)) (T5 C4 (EAST)))
     ((T3 C2 (EAST)) (T4 C4 (EAST))))
 (C4 (EAST) ((T4 C3 (EAST))) ((T4 C3 (EAST)) (T5 C3 (EAST))))
 (C5 (EAST)))
```




```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-SPACE '(0 1)))

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

(((0 1))
 ((IN A) ((0 1) = 0)
         ((C (NAND 1) ((0 1) = 0)) (C (NAND 2) ((0 1) = 0))))
 ((IN B) ((0 1) = 1)
         ((C (NAND 1) ((0 1) = 1)) (C (NAND 2) ((0 1) = 1))))
 ((NAND 1) ((0 1) = 1)
           ((C (NAND 2) ((0 1) = 1)) (C (NAND 3) ((0 1) = 1)))
           ((C (IN A) ((0 1) = 0)) (C (IN B) ((0 1) = 1))))
 ((NAND 2) ((0 1) = 1)
           ((C (NAND 4) ((0 1) = 1)))
           ((C (IN A) ((0 1) = 0))
            (C (IN B) ((0 1) = 1))
            (C (NAND 1) ((0 1) = 1))))
 ((NAND 3) ((0 1) = 0)
           ((C (NAND 4) ((0 1) = 0)))
           ((C (NAND 1) ((0 1) = 1))))
 ((NAND 4) ((0 1) = 1)
           ((C (OUT 1) ((0 1) = 1)))
           ((C (NAND 2) ((0 1) = 1)) (C (NAND 3) ((0 1) = 0))))
 ((OUT 1) ((0 1) = 1) NIL ((C (NAND 4) ((0 1) = 1)))))
```




```
?(PROGN (PRINT-SWITCHES) (DESCRIBE-SPACE '(0 1)))

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

(((0 1))
 ((IN A) ((0 1)))
 ((IN B) ((0 1)))
 ((NAND 1) ((0 1))
          NIL
          ((C (IN A) ((0 1) = 0)) (C (IN B) ((0 1) = 1))))
 ((NAND 2) ((0 1))
          NIL
          ((C (IN A) ((0 1) = 0))
           (C (IN B) ((0 1) = 1))
           (C (NAND 1) ((0 1) = 1))))
 ((NAND 3) ((0 1)) NIL ((C (NAND 1) ((0 1) = 1))))
 ((NAND 4) ((0 1))
          NIL
          ((C (NAND 2) ((0 1) = 1)) (C (NAND 3) ((0 1) = 0))))
 ((OUT 1) ((0 1)) NIL ((C (NAND 4) ((0 1) = 1)))))
```

```
?(DESCRIBE-SPACE '(X X))
*** DESCRIBE-SPACE ERROR: (X X) IS NOT A SPACE
```

(DESTROY-GRAPH)

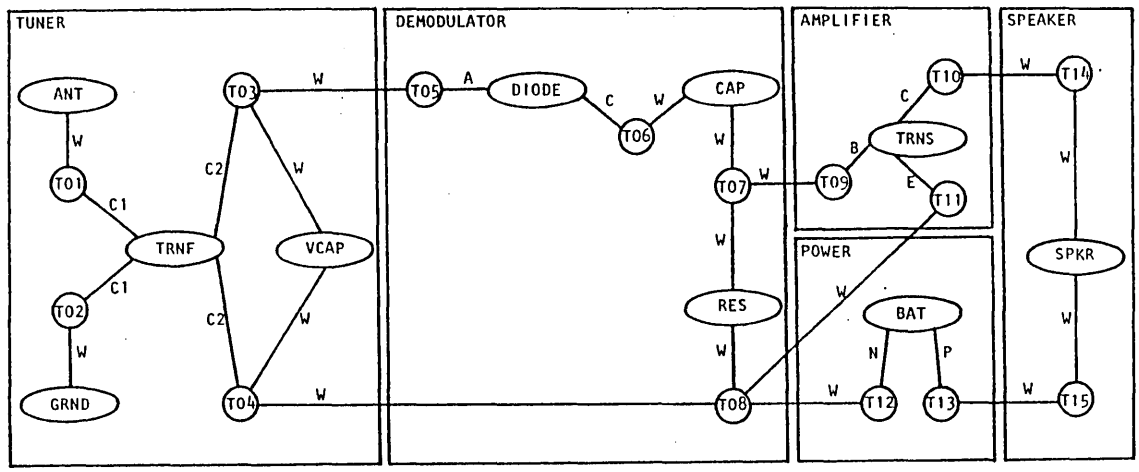
Informal Definition

The pseudo-function DESTROY-GRAPH is an EXPR which has the effect of destroying the existing GRAPH. DESTROY-GRAPH destroys all edges, nodes, and spaces except the universal space, UNIVERSE, which is rebound to NIL. DESTROY-GRAPH has no effect if the GRAPH is already empty and UNIVERSE is bound to NIL. DESTROY-GRAPH returns T.

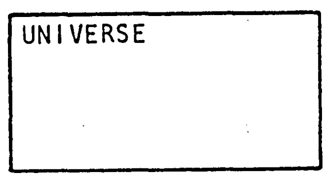
Formal Definition

DESTROY-GRAPH[ ] = T
with effects:
for each $s \in S$
DUS[s]

Illustrations



?(DESTROY-GRAPH)
T



(PRINT-GRAPH)

Informal Definition

The pseudo-function PRINT-GRAPH is an EXPR which "pretty prints" a GRAPH-DESCRIPTOR describing the existing GRAPH. PRINT-GRAPH does not print NIL values, descriptions of universal inclusion when the universal value is NIL, or descriptions of entities whose corresponding switches are off. The following format is used by PRINT-GRAPH. PRINT-GRAPH returns T.

```
<< space >> = space-value
      :            :
   node
     space = node-value-in-space
       :              :
        --outpointing-edge-> outpointing-node
          space = outpointing-pair-value-in-space
            :                     :
                  :            :
        <-inpointing-edge- inpointing-node
          space = inpointing-pair-value-in-space
            :                     :
                  :            :
              :                       :
```

Formal Definition

PRINT-GRAPH[ ] = T

with effects:

if SWITCH-S ≠ NIL

then for each $s_i \in$ S

PRINT[ ]

PRINT[ ]

if SWITCH-SV ≠ NIL and VUS[s_i] ≠ NIL

then PRINT[<<, s_i, >>, =, VUS[s_i]]

if s_i ≠ UNIVERSE

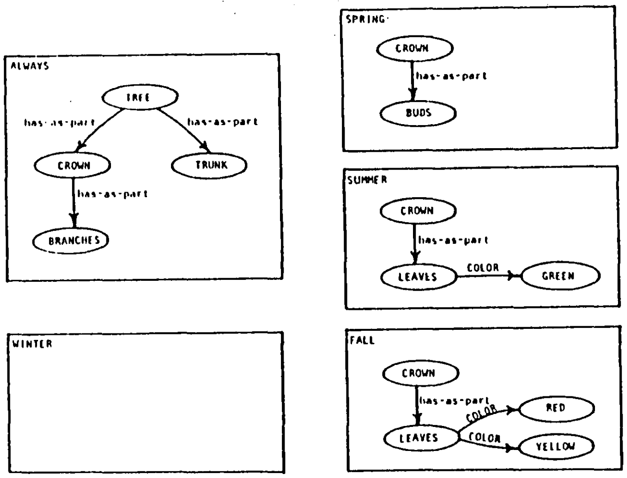
then PRINT[<<, s_i, >>]

if SWITCH-N ≠ NIL

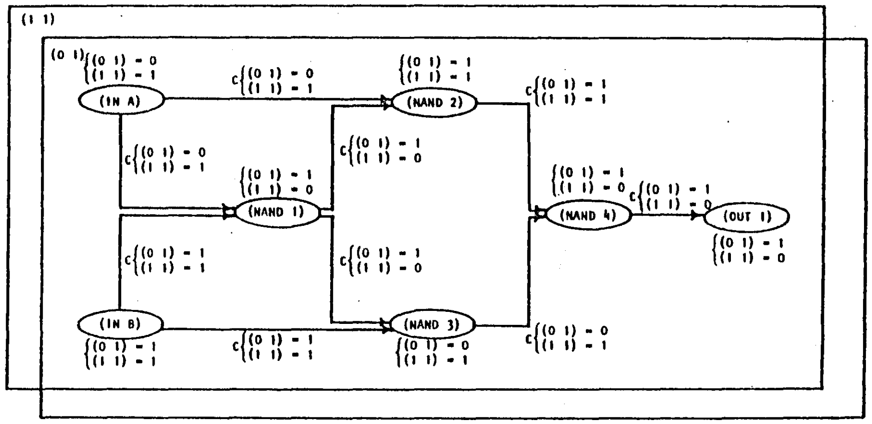
then for each $n_j \in$ N

PRINT-NODE[n_j, S]

Illustrations



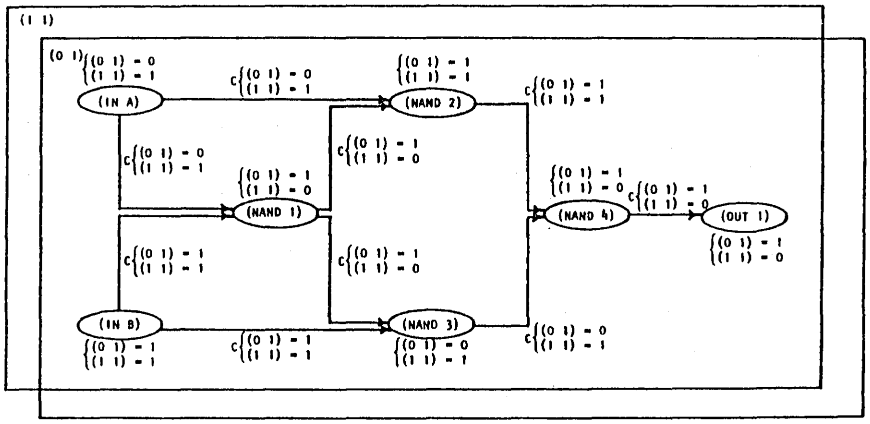

```
?(PROGN (PRINT-SWITCHES) (PRINT-GRAPH))

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

 << ALWAYS >>
 << FALL >>
 << SPRING >>
 << SUMMER >>
 << WINTER >>

 BRANCHES
   ALWAYS

     <-HAS-AS-PART-- CROWN
       ALWAYS

 BUDS
   SPRING

     <-HAS-AS-PART-- CROWN
       SPRING

 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

 GREEN
   SUMMER

     <-COLOR-- LEAVES
       SUMMER

 LEAVES
   FALL
   SUMMER

     --COLOR-> GREEN
       SUMMER

     --COLOR-> RED
       FALL

     --COLOR-> YELLOW
       FALL

     <-HAS-AS-PART-- CROWN
       FALL
       SUMMER

 RED
   FALL

     <-COLOR-- LEAVES
       FALL

 TREE
   ALWAYS

     --HAS-AS-PART-> CROWN
       ALWAYS

     --HAS-AS-PART-> TRUNK
       ALWAYS

 TRUNK
   ALWAYS

     <-HAS-AS-PART-- TREE
       ALWAYS

 YELLOW
   FALL

     <-COLOR-- LEAVES
       FALL
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-GRAPH))
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
 << (0 1) >>
 << (1 1) >>
  (IN A)
    (0 1) = 0
    (1 1) = 1
      --C-> (NAND 1)
        (0 1) = 0
        (1 1) = 1
      --C-> (NAND 2)
        (0 1) = 0
        (1 1) = 1
  (IN B)
    (0 1) = 1
    (1 1) = 1
      --C-> (NAND 1)
        (0 1) = 1
        (1 1) = 1
      --C-> (NAND 2)
        (0 1) = 1
        (1 1) = 1
  (NAND 1)
    (0 1) = 1
    (1 1) = 0
      --C-> (NAND 2)
        (0 1) = 1
        (1 1) = 0
      --C-> (NAND 3)
        (0 1) = 1
        (1 1) = 0
```

-- continued on next page --

```
          <-C-- (IN A)
            (0 1) = 0
            (1 1) = 1

          <-C-- (IN B)
            (0 1) = 1
            (1 1) = 1

      (NAND 2)
        (0 1) = 1
        (1 1) = 1

          --C-> (NAND 4)
            (0 1) = 1
            (1 1) = 1

          <-C-- (IN A)
            (0 1) = 1
            (1 1) = 1

          <-C-- (IN B)
            (0 1) = 1
            (1 1) = 1

          <-C-- (NAND 1)
            (0 1) = 1
            (1 1) = 0

      (NAND 3)
        (0 1) = 0
        (1 1) = 1

          --C-> (NAND 4)
            (0 1) = 0
            (1 1) = 1

          <-C-- (NAND 1)
            (0 1) = 1
            (1 1) = 0

      (NAND 4)
        (0 1) = 1
        (1 1) = 0

          --C-> (OUT 1)
            (0 1) = 1
            (1 1) = 0

          <-C-- (NAND 2)
            (0 1) = 1
            (1 1) = 1

          <-C-- (NAND 3)
            (0 1) = 0
            (1 1) = 1

      (OUT 1)
        (0 1) = 1
        (1 1) = 0

          <-C-- (NAND 4)
            (0 1) = 1
            (1 1) = 0
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-GRAPH))

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

  (IN A)
    (0 1) = 0
    (1 1) = 1

      --C-> (NAND 1)
        (0 1)
        (1 1)

      --C-> (NAND 2)
        (0 1)
        (1 1)

  (IN B)
    (0 1) = 1
    (1 1) = 1

      --C-> (NAND 1)
        (0 1)
        (1 1)

      --C-> (NAND 2)
        (0 1)
        (1 1)

  (NAND 1)
    (0 1) = 1
    (1 1) = 0

      --C-> (NAND 2)
        (0 1)
        (1 1)

      --C-> (NAND 3)
        (0 1)
        (1 1)

      <-C-- (IN A)
        (0 1)
        (1 1)

      <-C-- (IN B)
        (0 1)
        (1 1)
```

-- continued on next page --

```
(NAND 2)
  (0 1) = 1
  (1 1) = 1
    --C-> (NAND 4)
      (0 1)
      (1 1)
    <-C-- (IN A)
      (0 1)
      (1 1)
    <-C-- (IN B)
      (0 1)
      (1 1)
    <-C-- (NAND 1)
      (0 1)
      (1 1)
(NAND 3)
  (0 1) = 0
  (1 1) = 1
    --C-> (NAND 4)
      (0 1)
      (1 1)
    <-C-- (NAND 1)
      (0 1)
      (1 1)
(NAND 4)
  (0 1) = 1
  (1 1) = 0
    --C-> (OUT 1)
      (0 1)
      (1 1)
    <-C-- (NAND 2)
      (0 1)
      (1 1)
    <-C-- (NAND 3)
      (0 1)
      (1 1)
(OUT 1)
  (0 1) = 1
  (1 1) = 0
    <-C-- (NAND 4)
      (0 1)
      (1 1)
T
```

(PRINT-NODE *node*)[1]

Informal Definition

The pseudo-function PRINT-NODE is an EXPR which "pretty prints" a NODE-DESCRIPTOR for *node* in the existing GRAPH. Given *node*, PRINT-NODE prints a NODE-DESCRIPTOR describing *node* including information about all the spaces *node* is in. PRINT-NODE does not print NIL values, descriptions of universal inclusion when the universal value is NIL, or descriptions of those entities whose corresponding switches are off. The following format is used by PRINT-NODE. PRINT-NODE returns *node*.

```
node
  space = node-value-in-space
    :              :
     --outpointing-edge-> outpointing-node
       space = outpointing-pair-value-in-space
         :                     :
                  :            :
     <-inpointing-edge-- inpointing-node
       space = inpointing-pair-value-in-space
         :                     :
                  :            :
```

error condition:

- *node* does not exist

Formal Definition

PRINT-NODE[n] = n

with effects:

PRINT-NODE[n, S]

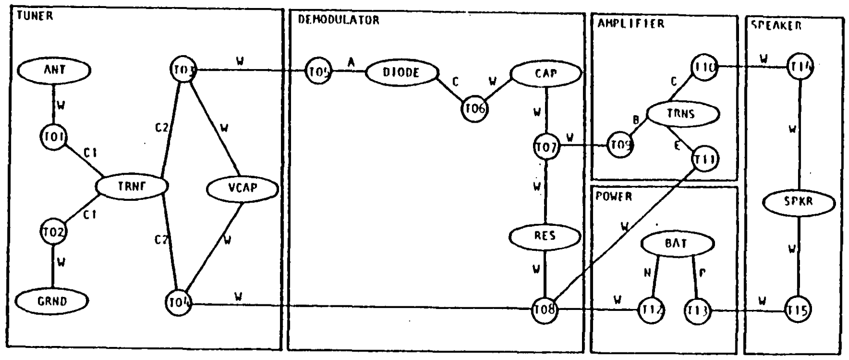
error condition:

- $n \notin N$

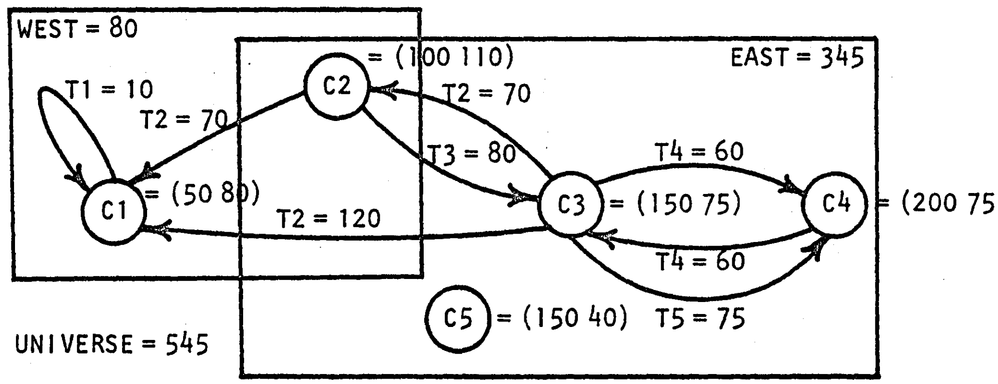
---

[1]See alternative form on page 306.

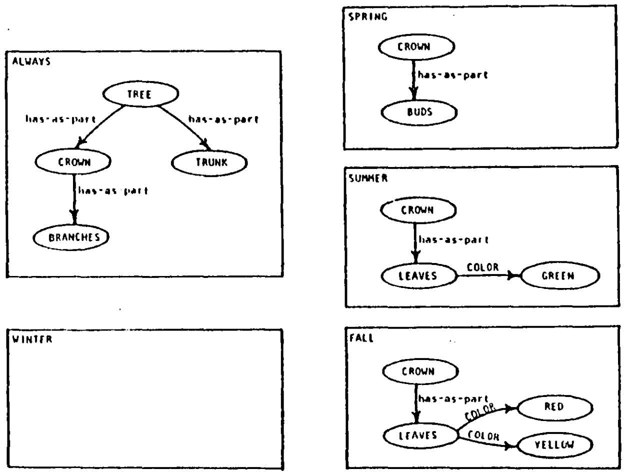
Illustrations




```
?(PROGN (PRINT-SWITCHES) (PRINT-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
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-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
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-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
T
?(PROGN (PRINT-SWITCHES) (PRINT-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

      --HAS-AS-PART-> BRANCHES

      --HAS-AS-PART-> BUDS

      --HAS-AS-PART-> LEAVES

      <-HAS-AS-PART-- TREE
T
?(PRINT-NODE 'NX)
*** PRINT-NODE ERROR: NX IS NOT A NODE
```

(PRINT-NODE *node list-of-spaces*)[1]

Informal Definition

The pseudo-function PRINT-NODE is an EXPR which "pretty prints" a NODE-DESCRIPTOR for *node* including information about the spaces in *list-of-spaces*. Given *node* and *list-of-spaces*, PRINT-NODE prints a NODE-DESCRIPTOR describing *node* including space information restricted to spaces in *list-of-spaces*. PRINT-NODE does not print NIL values, descriptions of universal inclusion when the universal value is NIL, or descriptions of those entities whose corresponding switches are off. The following format is used by PRINT-NODE. PRINT-NODE returns *node*.

```
node
  space = node-value-in-space
    :              :
    --outpointing-edge-> outpointing-node
      space = outpointing-pair-value-in-space
        :                    :
                 :           :
    <-inpointing-edge-- inpointing-node
      space = inpointing-pair-value-in-space
        :                    :
                 :           :
```

error conditions:

- *node* does not exist in any space in *list-of-spaces*
- some space in *list-of-spaces* does not exist

[1]See alternative form on page 302.

Formal Definition

```
PRINT-NODE[n, sl] = n
   with effects:
      PRINT[ ]
      PRINT[ ]
      SET-LEFT-MARGIN[2]
      PRINT[n]
      if SWITCH-NS ≠ NIL
         then print-nsv[n, sl]
      if SWITCH-OP ≠ NIL
         then print-op[n, sl]
      if SWITCH-IP ≠ NIL
         then print-ip[n, sl]
```

```
print-nsv[n, sl]
   with effects:
      SET-LEFT-MARGIN[ 4]
      where sl = (s_1 ... s_i ... s_t)
         for each s_i
            if XUN[n, s_i]
               then if SWITCH-NV ≠ NIL and VUN[n, s_i] ≠ NIL
                       then PRINT[s_i, =, VUN[n, s_i]]
                       else if s_i ≠ UNIVERSE
                               then PRINT[s_i]
```

```
print-op[n, sl]
   with effects:
      PRINT[ ]
      SET-LEFT-MARGIN[6]
      for each (g_i  m_i) ∈ SOP[n]
         where sl = (s_i ... s_t ... s_u)
            if for some s_t XOP[n, g_i, m_i, s_t] = T
               then PRINT[--g_i->, m_i]
                    if SWITCH-OPS ≠ NIL
                       then print-opsv[n, g, m, sl]
                    PRINT[ ]
```

-- continued on next page --

```
print-opsv[n, g, m, sl]
   with effects:
      SET-LEFT-MARGIN[8]
      where sl = (s_1 ... s_i ... s_t)
         if XOP[n, g, m, s_i] ≠ NIL
            then if SWITCH-OPV ≠ NIL and VOP[n, g, m, s_i] ≠ NIL
                    then PRINT[s_i, =, VOP[n, g, m, s_i]]
                    else if s_i ≠ UNIVERSE
                            then PRINT[s_i]
```

```
print-ip[n, sl]
   with effects:
      PRINT[ ]
      SET-LEFT-MARGIN[6]
      for each (g_i, m_i) ∈ SIP[n]
         where sl = (s_1 ... s_t ... s_u)
            if for some s_t XIP[n, g_i, m_i, s_t] = T
               then PRINT[<-g_i--, m_i]
                    if SWITCH-IPS ≠ NIL
                       then print-ipsv[n, g, m, sl]
                    PRINT[ ]
```

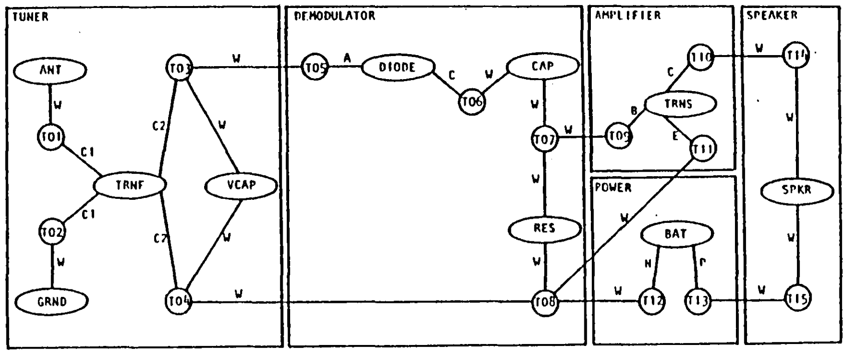
```
print-ipsv[n, g, m, sl]
   with effects:
      SET-LEFT-MARGIN[8]
      where sl = (s_1 ... s_i ... s_t)
         if XIP[n, g, m, s_i] ≠ NIL
            then if SWITCH-IPV ≠ NIL and VIP[n, g, m, s_i] ≠ NIL
                    then PRINT[s_i, =, VIP[n, g, m, s_i]]
                    else if s_i ≠ UNIVERSE
                            then PRINT[s_i]
```

---

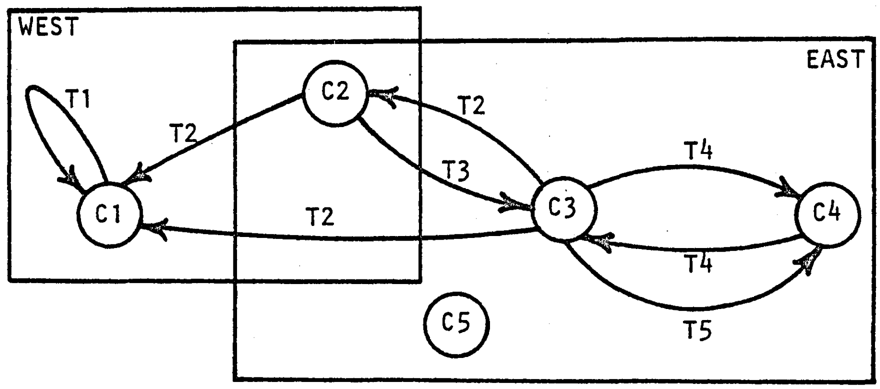
error conditions:

- $((n\ s_i)\ v) \notin NSV$ for all $v \in V$ where $sl = (s_1 \ldots s_i \ldots s_t)$
- some $s_i \notin S$ where $sl = (s_1 \ldots s_i \ldots s_t)$

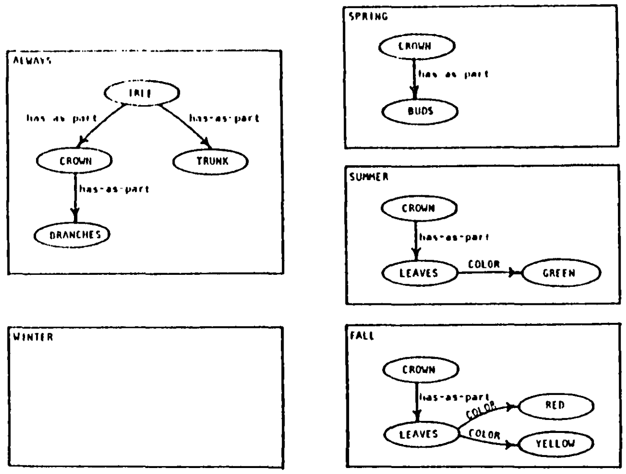
Illustrations




```
?(PROGN (PRINT-SWITCHES) (PRINT-NODE 'T04 '(TUNER POWER)))

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-> VCAP
        TUNER
      <-C2-- TRNF
        TUNER
      <-W-- VCAP
        TUNER
T
```




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

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

      --T3-> C3
        EAST

      <-T2-- C3
        EAST
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-NODE 'CROWN '(ALWAYS SUMMER)))

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
    SUMMER

      --HAS-AS-PART-> BRANCHES
        ALWAYS

      --HAS-AS-PART-> LEAVES
        SUMMER

      <-HAS-AS-PART-- TREE
        ALWAYS
T

?(PROGN (PRINT-SWITCHES) (PRINT-NODE 'CROWN '(ALWAYS SUMMER)))

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 = NIL          SWITCH-IPS = T             SWITCH-IPV = T

  CROWN

      --HAS-AS-PART-> BRANCHES

      --HAS-AS-PART-> LEAVES
T

?(PRINT-NODE 'LEAVES '(WINTER SPRING))

*** PRINT-NODE ERROR: LEAVES IS NOT A NODE IN SPACE (VIRTUAL-SPACE
                      (WINTER SPRING))[1]

?(PRINT-NODE 'LEAVES '(SUMMER SX))

*** PRINT-NODE ERROR: SX IS NOT A SPACE
```

[1] See virtual spaces on page 321.

(PRINT-SPACE *space*)

Informal Definition

The pseudo-function PRINT-SPACE is an EXPR which "pretty prints" a GRAPH-DESCRIPTOR for *space* in the existing GRAPH. Given *space*, PRINT-SPACE prints a description of *space* including a NODE-DESCRIPTOR for each node in *space* with information restricted to *space*. PRINT-SPACE does not print NIL values, descriptions of universal inclusion when the universal value is NIL, or descriptions of entities whose corresponding switches are off. The following format is used by PRINT-SPACE. PRINT-SPACE returns *space*.

```
<< space >> = space-value
  node = node-value-in-space
      --outpointing-edge-> outpointing-node = pair-value-in-space
                :                  :
      <-inpointing-edge- inpointing-node = pair-value-in-space
                :                  :
                    :                      :
```

error conditions:

- *space* does not exist

Formal Definition

```
PRINT-SPACE[s] = s
   with effects:
      if SWITCH-S ≠ NIL
         then PRINT[ ]
              PRINT[ ]
              if SWITCH-SV ≠ NIL and VUS[s] ≠ NIL
                 then PRINT[<<, s, >>, =, VUS[s]]
                 else PRINT[<<, s, >>]
      if SWITCH-N ≠ NIL
         then for each n_i ∈ SUN[s]
                 print-n[n_i, s]
```

```
print-n[n, s]
   with effects:
      PRINT[ ]
      PRINT[ ]
      SET-LEFT-MARGIN[2]
      if SWITCH-NS ≠ NIL and SWITCH-NV ≠ NIL and VUN[n, s] ≠ NIL
         then PRINT[n, =, VUN[n, s]]
         else PRINT[n]
      if SWITCH-OP ≠ NIL
         then print-op[n, s]
      if SWITCH-IP ≠ NIL
         then print-ip[n, s]
```

```
print-op[n, s]
   with effects:
      PRINT[ ]
      SET-LEFT-MARGIN[6]
      for each (g_i  m_i) ∈ SOP[n, s]
         if SWITCH-OPS ≠ NIL and SWITCH-OPV ≠ NIL and VOP[n, g_i, m_i, s] ≠ NIL
            then PRINT[--g_i->, m_i, =, VOP[n, g_i, m_i, s]]
                 PRINT[ ]
            else PRINT[--g_i->, m_i]
                 PRINT[ ]
```

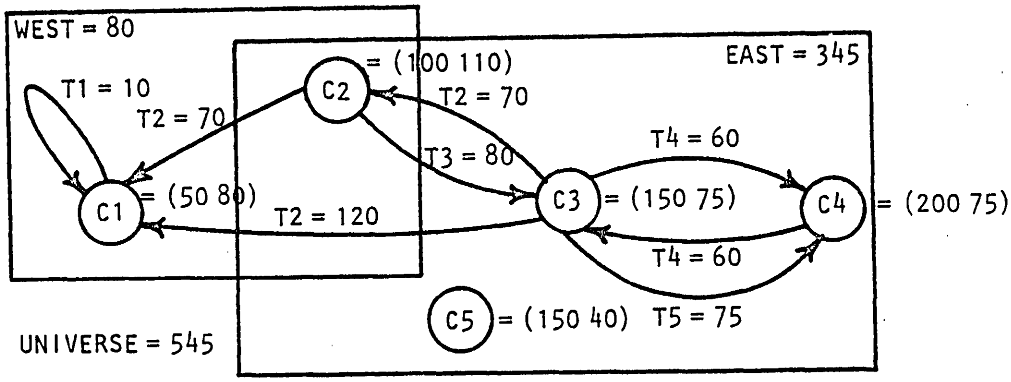
-- continued on next page --

```
print-ip[n, s]
   with effects:
      PRINT[ ]
      SET-LEFT-MARGIN[6]
      for each (g_i  m_i) ∈ SIP[n, s]
         if SWITCH-IPS ≠ NIL and SWITCH-IPV ≠ NIL and VIP[n, g_i, m_i, s] ≠ NIL
            then PRINT[<-g_i--, m_i, =, VIP[n, g_i, m_i, s]]
                 PRINT[ ]
            else PRINT[<-g_i--, m_i]
                 PRINT[ ]
```

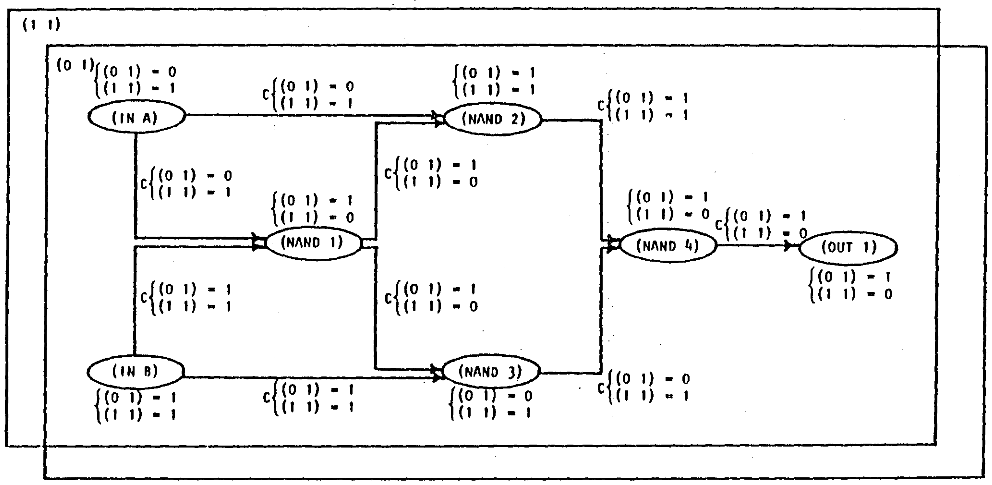
error condition:

- $s \notin S$

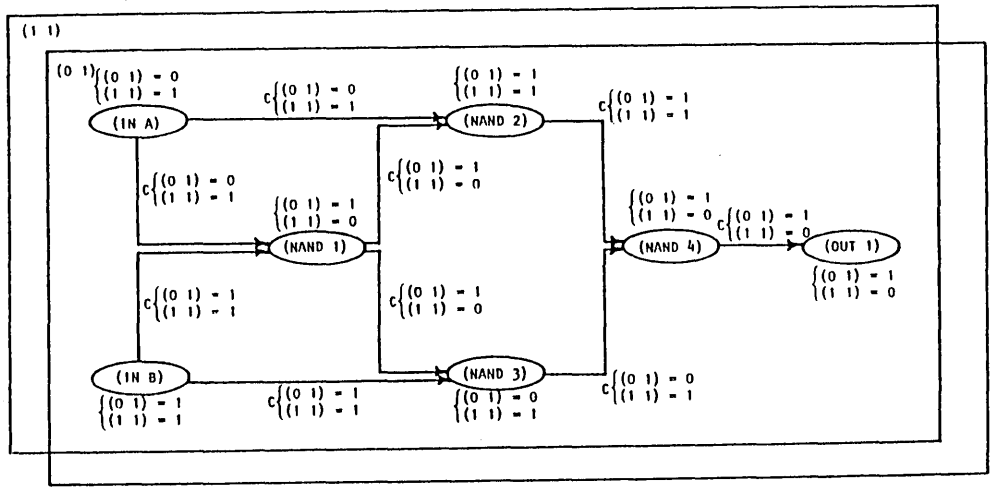
Illustrations




```
?(PROGN (PRINT-SWITCHES) (PRINT-SPACE 'EAST))

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

 << EAST >> = 345
  C2
      --T3-> C3
      <-T2-- C3
  C3
      --T2-> C2
      --T4-> C4
      --T5-> C4
      <-T3-- C2
      <-T4-- C4
  C4
      --T4-> C3
      <-T4-- C3
      <-T5-- C3
  C5
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-SPACE '(0 1)))
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
 << (0 1) >>
  (IN A) = 0
      --C-> (NAND 1) = 0
      --C-> (NAND 2) = 0
  (IN B) = 1
      --C-> (NAND 1) = 1
      --C-> (NAND 2) = 1
  (NAND 1) = 1
      --C-> (NAND 2) = 1
      --C-> (NAND 3) = 1
      <-C-- (IN A) = 0
      <-C-- (IN B) = 1
  (NAND 2) = 1
      --C-> (NAND 4) = 1
      <-C-- (IN A) = 0
      <-C-- (IN B) = 1
      <-C-- (NAND 1) = 1
  (NAND 3) = 0
      --C-> (NAND 4) = 0
      <-C-- (NAND 1) = 1
  (NAND 4) = 1
      --C-> (OUT 1) = 1
      <-C-- (NAND 2) = 1
      <-C-- (NAND 3) = 0
  (OUT 1) = 1
      <-C-- (NAND 4) = 1
T
```




```
?(PROGN (PRINT-SWITCHES) (PRINT-SPACE '(0 1)))
SWITCH-S = T                                             SWITCH-SV = T
SWITCH-N = T              SWITCH-NS = T                  SWITCH-NV = NIL
SWITCH-OP = NIL           SWITCH-OPS = T                 SWITCH-OPV = T
SWITCH-IP = T             SWITCH-IPS = T                 SWITCH-IPV = T
<< (0 1) >>
  (IN A)
  (IN B)
  (NAND 1)
      <-C-- (IN A) = 0
      <-C-- (IN B) = 1
  (NAND 2)
      <-C-- (IN A) = 0
      <-C-- (IN B) = 1
      <-C-- (NAND 1) = 1
  (NAND 3)
      <-C-- (NAND 1) = 1
  (NAND 4)
      <-C-- (NAND 2) = 1
      <-C-- (NAND 3) = 0
  (OUT 1)
      <-C-- (NAND 4) = 1
T
?(PRINT-SPACE '(X X))
*** PRINT-SPACE ERROR: (X X) IS NOT A SPACE
```

(PRINT-SWITCHES)

Informal Definition

The pseudo-function PRINT-SWITCHES is an EXPR which prints the Group II switches and their current values. The following format is used by PRINT-SWITCHES. PRINT-SWITCHES returns T.

0	24	50
SWITCH-S = SWITCH-S-value		SWITCH-SV = SWITCH-SV-value
SWITCH-N = SWITCH-N-value	SWITCH-NS = SWITCH-NS-value	SWITCH-NV = SWITCH-NV-value
SWITCH-OP = SWITCH-OP-value	SWITCH-OPS = SWITCH-OPS-value	SWITCH-OPV = SWITCH-OPV-value
SWITCH-IP = SWITCH-IP-value	SWITCH-IPS = SWITCH-IPS-value	SWITCH-IPV = SWITCH-IPV-value

Formal Definition[1,2]

PRINT-SWITCHES[ ] = T
with effects:
PRINT[ ]
PRINT['SWITCH-S, =, SWITCH-S, 'SWITCH-SV^{50}, =, SWITCH-SV]
PRINT['SWITCH-N, =, SWITCH-N, 'SWITCH-NS^{24}, =, SWITCH-NS,
'SWITCH-NV^{50}, =, SWITCH-NV]
PRINT['SWITCH-OP, =, SWITCH-OP, 'SWITCH-OPS^{24}, =, SWITCH-OPS,
'SWITCH-OPV^{50}, =, SWITCH-OPV]
PRINT['SWITCH-IP, =, SWITCH-IP, 'SWITCH-IPS^{24}, =, SWITCH-IPS,
'SWITCH-IPV^{50}, =, SWITCH-IPV]
PRINT[ ]

[1] Quoted arguments to PRINT indicate that the argument is to be printed, not its value.

[2] Superscripts on arguments to PRINT indicate the position the carriage is to move to before printing that argument.

Illustrations

```
?(PRINT-SWITCHES)

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

T
```

Group III Primitives

Group III primitives pertain to memory management. They control the GRASPER virtual memory system for GRAPH storage, move GRAPHs in and out of long-term storage, and provide a means of specifying a subset of spaces that are to be treated as a single space by GRASPER operators.

Virtual Spaces

Spaces and virtual spaces are both mechanisms for specifying subgraphs. Spaces are real entities that are created, inspected, and destroyed. Virtual spaces are not real entities. They are neither created nor destroyed; they can only be inspected.

A virtual space is a view through a set of spaces possibly including other virtual spaces. A virtual space contains exactly those entities contained in at least one of the spaces it is defined over. This does not include the values of the entities in those spaces.

All Group I "S" and "X" operators and Group II "DESCRIBE" and "PRINT" operators that have a space as an argument can be given a virtual space instead. The result of these operators will be the same as if a real space had been given to the operators containing exactly those entities in the virtual space. An error will result if any of the other Group I or Group II operators are called with a virtual space in place of a real space.

A description of each real space is stored in memory. The description of a virtual space is not stored but dynamically determined each time it is referenced. Thus, virtual spaces are more memory efficient but have slower access times than real spaces.

A virtual space is defined by a list of two elements. The first element is the atom VIRTUAL-SPACE. The second element is the list of spaces (possibly including other virtual spaces) it is defined over. The following S-expression defines a virtual space over spaces s_1 through s_n.

$$\text{(VIRTUAL-SPACE } (s_1\ s_2\ \ldots\ s_n))$$

The function VIRTUAL-SPACE returns such a definition when given a list of spaces.

Virtual UNIVERSE

The universal space is normally maintained by GRASPER as a real space. 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 utilized to store the GRAPH is only about half of that necessary to store the GRAPH in real-UNIVERSE mode. Updates to the GRAPH are faster in virtual-UNIVERSE mode since UNIVERSE need not be updated. But as with all virtual spaces, access times are slower when referencing UNIVERSE in virtual-UNIVERSE mode since the description of UNIVERSE must be dynamically determined.

VIRTUALIZE-UNIVERSE and REALIZE-UNIVERSE are the operators that switch GRASPER between virtual-UNIVERSE and real-UNIVERSE modes. The operators INPUT-GRAPH and RESET may also cause the mode to switch. The Group I operator XUS can be used to determine the current mode of UNIVERSE.

Virtual Memory Management

GRASPER supports a virtual memory system for GRAPH storage. GRAPHs that are too large to be stored in primary memory are partitioned into pages and stored in secondary memory. A page remains in secondary memory until a GRASPER operator references its contents. At that time, the referenced page is moved into primary memory. If there is not sufficient space remaining in primary memory to accommodate that page, enough of the pages already in primary memory are moved back to secondary memory to accommodate it. Those pages least recently referenced are the first to be moved back to secondary memory.

Each space has at least one associated page. If a space is sufficiently large, it may have several associated pages. Each page contains descriptions of a subset of the nodes in that space. The user has control of the maximum PAGE size.[1] Whenever this maximum size is about to be exceeded, the page is split into two pages of approximately equal size. When pages of a single space become exceedingly small,[2] they are merged with other pages of that space. Appropriate splits and merges are performed whenever the maximum PAGE size is changed.

The maximum amount of primary memory used for GRAPH storage is controlled by the user. This maximum MEMORY size is the highest PAGE size total allowed in primary memory at once. The virtual memory manager guarantees that this maximum will not be exceeded even when it is dynamically varied.

The maximum PAGE and MEMORY sizes are initially infinite. This means that the entire GRAPH is in primary memory and each space is stored as a single page. The size settings can be changed by using the operator SET-SIZE. The operators INPUT-GRAPH and RESET may also affect these settings. Current sizes are returned by the operator SIZE.

---

[1] The PAGE size metric is implementation dependent.

[2] This minimum PAGE size is implementation dependent.

NUMBER Size

The number of significant digits in the coefficient of floating point numbers stored in GRAPHs is controlled by the setting of NUMBER size. It is initially set to 5. This setting can be changed by using the operator SET-SIZE. The operators INPUT-GRAPH and RESET may also affect this setting. The operator SIZE can be used to determine the current setting of NUMBER size.

Group III Operator Descriptions

This section contains a complete description of each Group III operator in alphabetical order. Each description consists of

1) the calling form (in LISP syntax) of the operator with its arguments,
2) its informal definition including
    (a) a prose description of the operator's purpose,
    and (b) a prose description of each GRASPER error condition,
3) its formal definition including all GRASPER error conditions,

and 4) a group of illustrations (in LISP syntax) including the generation of each GRASPER error condition.

Each group of illustrations begins with a drawing of the GRAPH which exists before each illustrative call. The series of calls does not represent a sequence during one user session. If a call alters the GRAPH, a drawing of the resulting GRAPH is given.

The formal definitions only relate to the GRAPH semantics. They do not specify states or changes in state of the implementing machine. For example, the transfer of information between primary and secondary memory only relates to the implementing machine; therefore, this is not included in the formal definitions.

The descriptions follow in alphabetical order.

(INPUT-GRAPH *file*)

Informal Definition

The pseudo-function INPUT-GRAPH is an EXPR which has the effect of inputing GRAPH from *file*.[1] Given *file*, INPUT-GRAPH replaces the current GRAPH with the GRAPH stored in *file*. INPUT-GRAPH resets NUMBER size, PAGE size, and the UNIVERSE mode to what they were when the GRAPH was output to *file* by OUTPUT-GRAPH. INPUT-GRAPH returns *file*.

error conditions:

- *file* is not a GRASPER file containing a GRASPER-GRAPH
- the PAGE size associated with *file* is greater than MEMORY size

Formal Definition

```
INPUT-GRAPH[f] = f
   with effects:
      GRASPER-GRAPH := GRAPH-IN-FILE[2][f]
      SET-SIZE[NUMBER, SIZE-OF[3][NUMBER, f]
               PAGE, SIZE-OF[PAGE, f]]
      If VIRTUAL-UNIVERSE?[4][f] = T
         then VIRTUALIZE-UNIVERSE[ ]
```

error condition:

- SIZE-OF[PAGE, f] > SIZE[MEMORY]

---

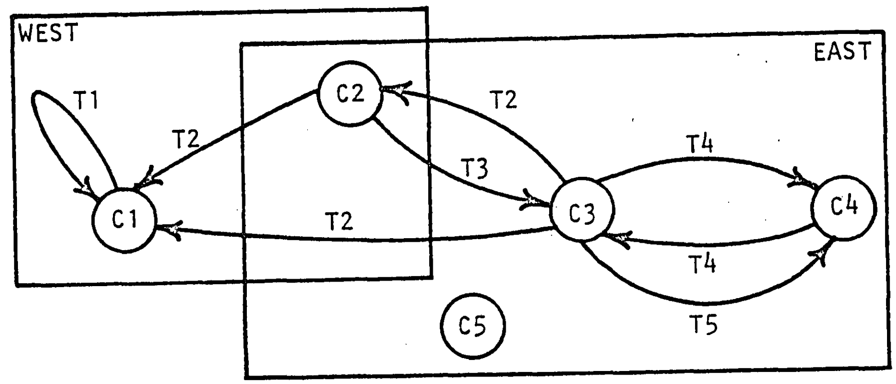
[1]The means of specifying a file is implementation dependent.

[2]GRAPH-IN-FILE returns the GRASPER-GRAPH last output to the given file.

[3]SIZE-OF returns the setting of the given size parameter when the GRAPH on the given file was output.

[4]VIRTUAL-UNIVERSE? returns T if the system was in virtual-UNIVERSE mode when the GRAPH on the given file was output; otherwise it returns NIL.

Illustrations




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

```
?(INPUT-GRAPH 'NOGRAPH)
*** INPUT-GRAPH ERROR: FILE  NOGRAPH IS NOT A GRASPER FILE
```

```
?(PROGN (SET-SIZE 'PAGE 50 'MEMORY 100)
        (OUTPUT-GRAPH 'FILE1)
        (RESET)
        (SET-SIZE 'MEMORY 25)
        (INPUT-GRAPH 'FILE1))
*** INPUT-GRAPH ERROR: PAGE SIZE CANNOT BE LARGER THAN MEMORY SIZE
                       THE CONFLICTING VALUES WERE:
                            MEMORY SIZE = 25
                            PAGE SIZE = 50
```

(OUTPUT-GRAPH *file*)

Informal Definition

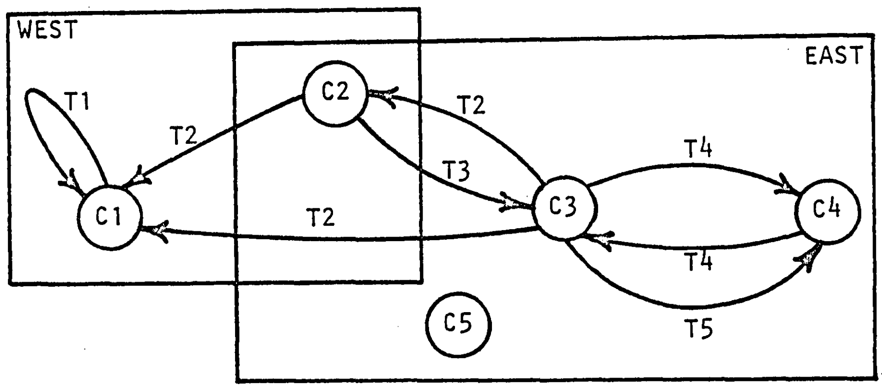
The pseudo-function OUTPUT-GRAPH is an EXPR which has the effect of outputing the current GRAPH to *file*.[1] If *file* does not already exist, OUTPUT-GRAPH creates it and stores GRAPH in it. If *file* does exist, its contents are replaced with GRAPH. OUTPUT-GRAPH returns *file*.

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
```

(REALIZE-UNIVERSE)

Informal Definition

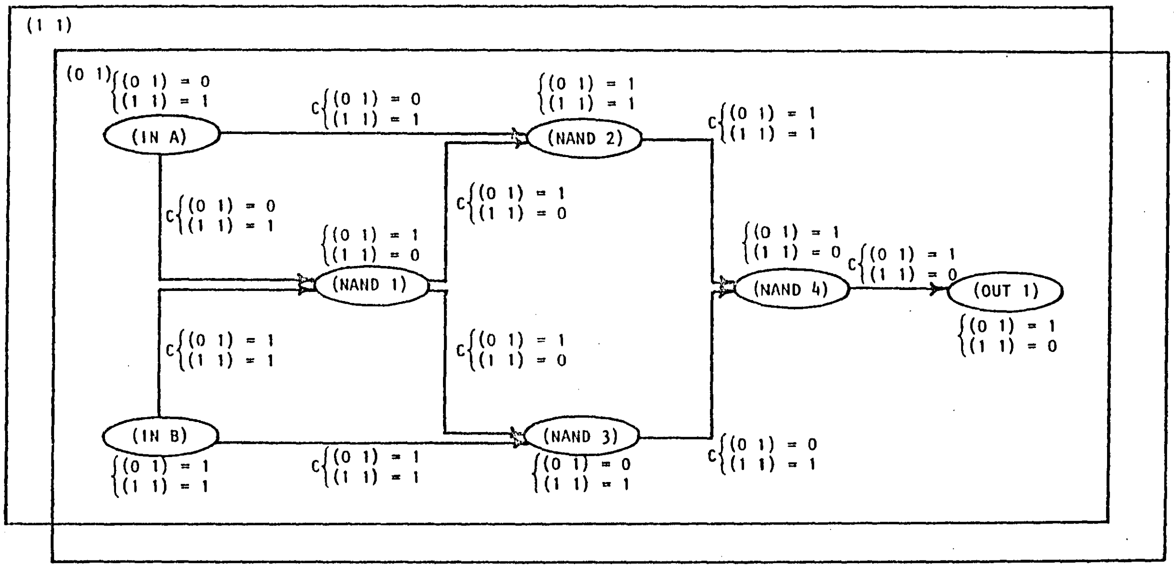
The pseudo-function REALIZE-UNIVERSE is an EXPR which puts GRASPER in real-UNIVERSE mode. If GRASPER is in virtual-UNIVERSE mode, REALIZE-UNIVERSE creates a real universal space containing all the nodes and edges contained in the other spaces. If GRASPER is in real-UNIVERSE mode, REALIZE-UNIVERSE has no effect. REALIZE-UNIVERSE returns T.

Formal Definition

REALIZE-UNIVERSE[ ] = T

All GRASPER operators will now reference GRASPER-GRAPH regardless of any previous calls to VIRTUALIZE-UNIVERSE.

Illustrations



?(REALIZE-UNIVERSE)
T

(RESET)

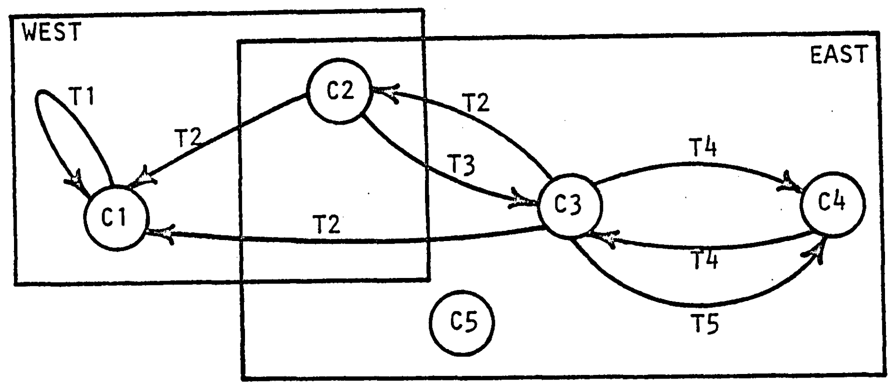
Informal Definition

The pseudo-function RESET is an EXPR which restores the system to its initial state. RESET destroys GRAPH, puts GRASPER in real-UNIVERSE mode, and resets NUMBER size to 5, PAGE size to infinity, MEMORY size to infinity, and all Group II switches on. RESET returns T.

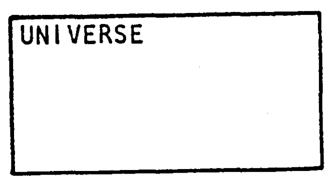
Formal Definition

RESET[ ] = T
  with effects:
    DESTROY-GRAPH[ ]
    REALIZE-UNIVERSE[ ]
    SET-SIZE[NUMBER, 5, PAGE, ∞, MEMORY, ∞]
    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

Illustrations



?(RESET)
T



(SET-SIZE *size-parameter*$_1$ *size*$_1$ ... *size-parameter*$_n$ *size*$_n$)

Informal Definition

The pseudo-function SET-SIZE is an EXPR which has the effect of setting *size-parameter*$_1$,...,*size-parameter*$_n$ to *size*$_1$,..., *size*$_n$. Given *size-parameter*$_1$, *size*$_1$,...,*size-parameter*$_n$, *size*$_n$, for each i, SET-SIZE sets *size-parameter*$_i$ to *size*$_i$. The setting for PAGE is the maximum PAGE size. The setting for MEMORY is the maximum PAGE size total in primary memory at one time. The setting for NUMBER is the number of significant digits in the coefficient of floating point numbers stored in GRAPH. Changing the setting of PAGE and/or MEMORY may cause the GRASPER memory manager to reconfigure the way GRAPH is stored.

error conditions:

- some *size-parameter*$_i$ is neither PAGE, MEMORY, nor NUMBER
- some *size*$_i$ is not a positive integer
- MEMORY size and/or PAGE size are set such that MEMORY size is less than PAGE size

Formal Definition

SET-SIZE[sp_1, s_1,...,sp_n, s_n] = T
with effects:
for each sp_i SET-SIZE-TO[1] [sp_i, s_i]

error conditions:

- some $sp_i \notin$ {PAGE, MEMORY, NUMBER}
- some $s_i \notin \{int | int \text{ is an integer}, int > 0\}$
- SIZE[MEMORY] < SIZE[PAGE] after execution

[1]SET-SIZE-TO sets the given size parameter to the given size.

Illustrations

```
?(SET-SIZE 'MEMORY 100 'PAGE 50 'NUMBER 5)
(MEMORY 100 PAGE 50 NUMBER 5)


?(PROGN (SET-SIZE 'MEMORY 100 'PAGE 50 'NUMBER 5)
        (SIZE 'PAGE 'MEMORY 'NUMBER))
(50 100 5)


?(PROGN (SET-SIZE 'MEMORY 100) (SIZE 'MEMORY))
(100)


?(SET-SIZE 'XXX 100)
*** SET-SIZE ERROR: XXX IS NOT A LEGAL ARGUMENT
                    THE LEGAL ARGUMENTS ARE:
                         1. NUMBER
                         2. MEMORY
                         3. PAGE


?(SET-SIZE 'MEMORY 'X)
*** SET-SIZE ERROR: MEMORY SIZE MUST BE A POSITIVE INTEGER
                    THE SIZE PROVIDED WAS X


?(SET-SIZE 'MEMORY 50 'PAGE 100)
*** SET-SIZE ERROR: PAGE SIZE CANNOT BE
                    LARGER THAN MEMORY SIZE
                    THE CONFLICTING VALUES WERE:
                         MEMORY SIZE = 50
                         PAGE SIZE   = 100
```

(SIZE *size-parameter*$_1$ ... *size-parameter*$_n$)

Informal Definition

The function SIZE is an EXPR which returns a list of the current settings of *size-parameter*$_1$,...,*size-parameter*$_n$. The setting for PAGE is the maximum PAGE size. The setting for MEMORY is the maximum PAGE size total in primary memory at one time. The setting for NUMBER is the number of significant digits in the coefficient of floating point numbers stored in GRAPH.

error condition:

- some *size-parameter*$_i$ is neither PAGE, MEMORY, nor NUMBER

Formal Definition

SIZE[$sp_1,\ldots,sp_n$] = (s_1 ... s_n)
where s_i = SETTING-OF[1][sp_i]

error condition:

- $sp_1 \notin$ {PAGE, MEMORY, NUMBER}

[1]SETTING-OF returns the current setting of the given size parameter.

Illustrations

```
?(PROGN (SET-SIZE 'MEMORY 100 'PAGE 50 'NUMBER 5)
        (SIZE 'PAGE 'MEMORY 'NUMBER))
(50 100 5)
```

```
?(PROGN (SET-SIZE 'MEMORY 100) (SIZE 'MEMORY))
(100)
```

```
?(SIZE 'XXX)
*** SIZE ERROR: XXX IS NOT A LEGAL ARGUMENT
                THE LEGAL ARGUMENTS ARE:
                    1. NUMBER
                    2. MEMORY
                    3. PAGE
```

(VIRTUAL-SPACE *list-of-spaces*)

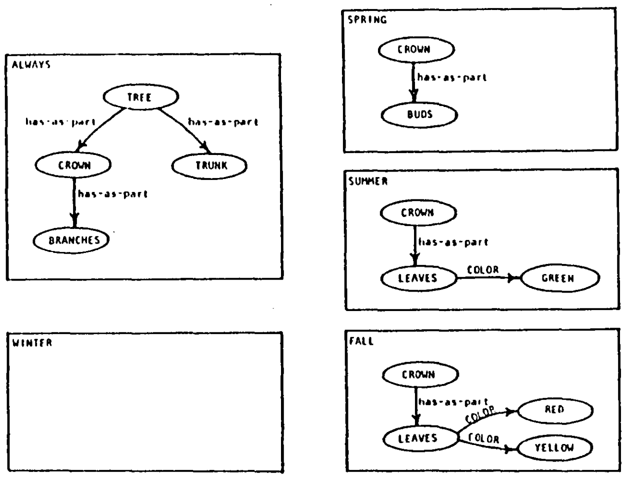
Informal Definition

The function VIRTUAL-SPACE is an EXPR which returns a definition of a virtual space defined over the spaces in *list-of-spaces*. Given *list-of-spaces*, VIRTUAL-SPACE returns a list of two elements. The first element is the atom VIRTUAL-SPACE and the second element is *list-of-spaces*. The resulting virtual space definition can be used as the space argument to Group I "S" and "X" operators and Group II "DESCRIBE" and "PRINT" operators.

Formal Definition

VIRTUAL-SPACE[ls] - (VIRTUAL-SPACE ls)

Illustrations




```
?(VIRTUAL-SPACE '(ALWAYS SUMMER))
(VIRTUAL-SPACE (ALWAYS SUMMER))

?(SUN (VIRTUAL-SPACE '(FALL SPRING WINTER)))
(BUDS CROWN LEAVES RED YELLOW)

?(PRINT-SPACE (VIRTUAL-SPACE '(ALWAYS SUMMER)))

<< (VIRTUAL-SPACE (ALWAYS SUMMER)) >>

 BRANCHES

     <-HAS-AS-PART-- CROWN

 CROWN

     --HAS-AS-PART-> BRANCHES
     --HAS-AS-PART-> LEAVES
     <-HAS-AS-PART-- TREE

 GREEN

     <-COLOR-- LEAVES

 LEAVES

     --COLOR-> GREEN
     <-HAS-AS-PART-- CROWN

 TREE

     --HAS-AS-PART-> CROWN
     --HAS-AS-PART-> TRUNK

 TRUNK

     <-HAS-AS-PART-- TREE
T
```

(VIRTUALIZE-UNIVERSE)

Informal Definition

The pseudo-function VIRTUALIZE-UNIVERSE is an EXPR which puts GRASPER in virtual-UNIVERSE mode. If GRASPER is in real-UNIVERSE mode, VIRTUALIZE-UNIVERSE destroys UNIVERSE and redefines it to be a virtual space over all other spaces. Nodes and edges that only existed in UNIVERSE are lost along with all universal values. If GRASPER is in virtual-UNIVERSE mode, VIRTUALIZE-UNIVERSE has no effect. VIRTUALIZE-UNIVERSE returns T.

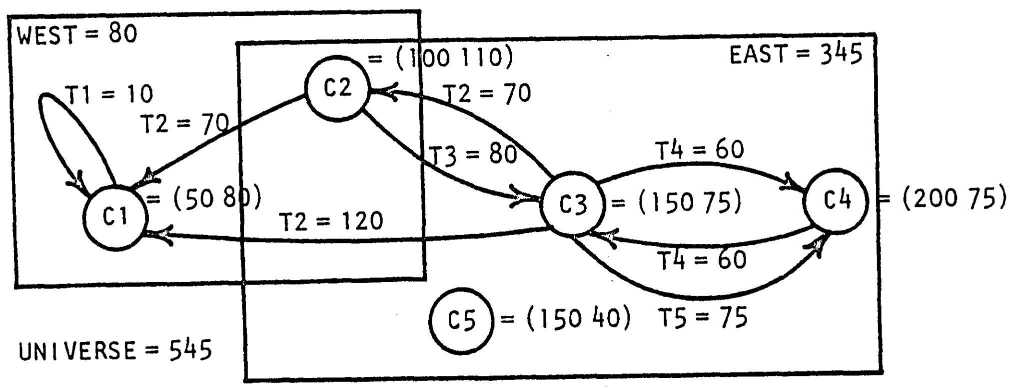
Formal Definition

```
VIRTUALIZE-UNIVERSE[ ] = T
   with effects:
      for each n ∈ {n|SUS[n] = NIL}
          DUN[n]
      for each (n g m) ∈ {(n g m)|(n g m) ∈ NGN and
                                  (((n g m) s) v) ∉ NGNSV for all
                                     s ∈ S - {UNIVERSE}, v ∈ V}
          DOP[ngm]
      BUS[UNIVERSE, NIL]
      for each n ∈ N
          BUN[n, NIL]
      for each (n g m) ∈ NGM
          BOP[n, g, m, NIL]
```

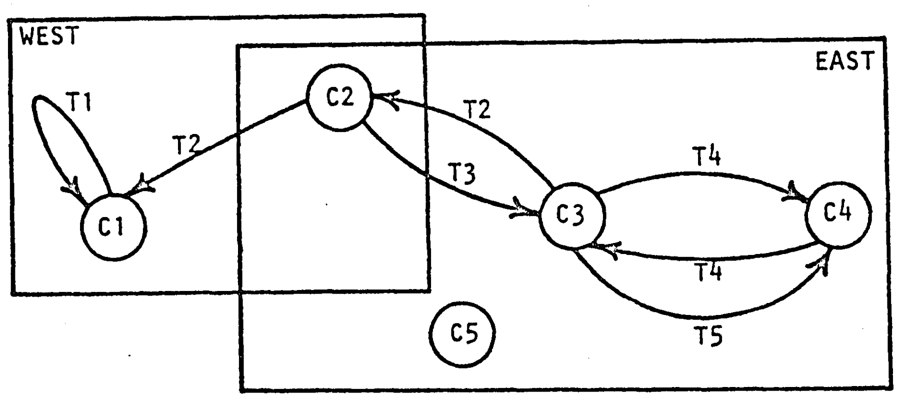
All GRASPER operators other than types "S", "X", "DESCRIBE", and "PRINT" will now reference GRASPER-GRAPH' instead of GRASPER-GRAPH.

```
     GRASPER-GRAPH' = (N' NGN' S' NSV' NGNSV' SV')
        where N' = N
              NGN' = NGN
              S' = S - {UNIVERSE}
              NSV' = NSV - {((n UNIVERSE) v)|v ∈ V}
              NGNSV' = NGNSV - {((((n g m) UNIVERSE) v)|v ∈ V}
              SV' = SV - {(UNIVERSE v)|v ∈ V}
```

Illustrations




```
?(VIRTUALIZE-UNIVERSE)
T
```



APPENDIX

Appendix A

Pronunciation Symbols [WEB70]

a ... mat, map, gag, snap

ä ... bother, cot

b ... baby, rib

d ... did, adder

ə ... banana, collect, abut

g ... go, big, gift

i ... tip, banish, active

k ... kin, cook, ache

n ... no, own

ŋ ... finger, ink, thing

ȯ ... saw, all, gnaw

p ... pepper, lip

s ... source, less

v ... vivid, give

z ... zone, raise, xylophone

\- ... mark of syllable division

' ... mark preceding the syllable with primary (strongest) stress

Appendix B

Implementations

A version of GRASPER 1.0 has been implemented and is operating on the CDC Cyber-74 installation at the University of Massachusetts Computing Center. ALISP [KON75], a version of LISP 1.5, serves as the host language. This system is a faithful implementation of the language described in this manual including the auxiliary operators described in Appendix C (pp. 345-377).

Another version based on LISP F3 [NOR78a, NOR78b] resides on the PDP VAX 11/780 of the Computer and Information Science Research Lab at the University of Massachusetts. It also is faithful to this manual including the auxiliary operators.

Anyone wishing to use these systems should first familiarize themselves with the respective version of LISP. The LISP News for each system details the loading procedure for GRASPER. Implementation dependent information is contained in the GRASPER News. The auxiliary operators PRINT-NEWS and PRINT-ALL-NEWS can be used to obtain a copy of the GRASPER News.

Appendix C

Auxiliary Operators

This appendix contains a complete description of each GRASPER auxiliary operator. Although these operators are not formally part of GRASPER, they (along with many of the operators already available in LISP) are needed to make GRASPER a viable language.

These auxiliary operators include the common set operators, a number of functional operators providing a more general mapping facility than is available in LISP [FRI74], and operators which print the GRASPER News. Each description consists of

1) the calling form (in LISP syntax) of the operator with its arguments,
2) its informal definition including
    (a) a prose description of the operator's purpose,
    and (b) a prose description of each GRASPER error condition,
3) its formal definition including all GRASPER error conditions,

and 4) a group of illustrations (in LISP syntax) including the generation of each error condition.

The descriptions follow in alphabetical order.

(ADD-ELEMENT *element set*)[1]

Informal Definition

The function ADD-ELEMENT is an EXPR which returns a set resulting from adding *element* to *set*. Given *element* and *set*, ADD-ELEMENT returns *set* if *element* is contained in *set*. If *element* is not in *set*, ADD-ELEMENT returns a set containing *element* and all the elements in *set*.

Formal Definition

ADD-ELEMENT[e,s] = UNION[(e),s]

[1] "Set" refers to a list with no EQUAL elements.

Illustrations

```
?(ADD-ELEMENT 'E2 '(E1 E3 E4 E5))
(E2 E1 E3 E4 E5)


?(ADD-ELEMENT 'E2 '(E1 E2 E3 E4 E5))
(E1 E2 E3 E4 E5)


?(ADD-ELEMENT 'E2 NIL)
(E2)


?(ADD-ELEMENT '(E 2) '((E 1) (E 3) (E 4) (E 5)))
((E 2) (E 1) (E 3) (E 4) (E 5))
```

(DIFFERENCE set_1 set_2)[1]

Informal Definition

The function DIFFERENCE is an EXPR which returns the difference of set_1 and set_2. Given set_1 and set_2, DIFFERENCE returns a set consisting of all elements that belong to set_1 but not to set_2.

Formal Definition

DIFFERENCE[s1,s2] = $(e_1 \ldots e_n)$

where s1 = $(e_{11} \ldots e_{1k})$

s2 = $(e_{21} \ldots e_{2m})$

$e_i \in \{e_{11},\ldots,e_{1k}\}$

$e_i \notin \{e_{21},\ldots,e_{2m}\}$

[1] "Set" refers to a list with no EQUAL elements.

Illustrations

```
?(DIFFERENCE '(E1 E2 E4 E5) '(E3 E5 E6))
(E1 E2 E4)


?(DIFFERENCE '(E1 E2 E4 E5) '(E3 E6 E7))
(E1 E2 E4 E5)


?(DIFFERENCE '(E1 E2 E4 E5) '(E1 E2 E3 E4 E5 E6))
NIL


?(DIFFERENCE '(E1 E2 E3) NIL)
(E1 E2 E3)


?(DIFFERENCE '((E 1) (E 2) (E 4) (E 5)) '((E 2) (E 4)))
((E 1) (E 5))
```

(ELEMENT? *element set*)[1]

Informal Definition

The function ELEMENT? is an EXPR which tests if *element* is in *set*. Given *element* and *set*, ELEMENT? returns T if *element* is in *set* and NIL if it is not.

Formal Definition

ELEMENT?[e,s] = r
where $s = \{e_1,\ldots,e_n\}$
if $e \in \{e_1,\ldots,e_n\}$
then r = T
else r = NIL

[1] "Set" refers to a list with no EQUAL elements.

Illustrations

```
?(ELEMENT 'E2 '(E1 E2 E3))
T
```

```
?(ELEMENT? 'E2 '(E1 E3 E4))
NIL
```

```
?(ELEMENT? 'E2 NIL)
NIL
```

```
?(ELEMENT? '(E 2) '((E 1) (E 2) (E 3)))
T
```

(FC *operator initial-value operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The function FC is an EXPR which uses *operator* to combine the results produced by applying *operator-list* to $arg\text{-}list_1,\ldots,arg\text{-}list_n$ as a functional combinator with *initial-value*. Given *operator*, *initial-value*, *operator-list*, $arg\text{-}list_1,\ldots,$ and $arg\text{-}list_n$, FC treats *operator-list*, $arg\text{-}list_1,\ldots,arg\text{-}list_n$ as successive rows of a matrix. Each row is truncated to the length of the shortest. Then the operator at the head of each column is APPLYed to the remainder of the column less any occurrences of the atom PSEUDO. FC then uses *operator* to successively combine these results and *initial-value*.

Formal Definition

$$
\begin{array}{l}
\mathrm{FC}[\mathrm{op},\ \mathrm{ival},\ \mathrm{ol},\ \mathrm{al}_1,\ \mathrm{al}_2,\ldots,\mathrm{al}_n] = \mathrm{r} \\
\quad \text{where } \mathrm{ol} = (\mathrm{o}_1\ \mathrm{o}_2\ \ldots\ \mathrm{o}_{t0}) \\
\qquad \mathrm{al}_1 = (\mathrm{a}_{11}\ \mathrm{a}_{12}\ \ldots\ \mathrm{a}_{1t1}) \\
\qquad \mathrm{al}_2 = (\mathrm{a}_{21}\ \mathrm{a}_{22}\ \ldots\ \mathrm{a}_{2t2}) \\
\qquad \quad \vdots \\
\qquad \mathrm{al}_n = (\mathrm{a}_{n1}\ \mathrm{a}_{n2}\ \ldots\ \mathrm{a}_{ntn}) \\
\quad \text{if } \mathrm{ol} = (\ ) \text{ or } \mathrm{al}_1 = (\ ) \text{ or } \mathrm{al}_2 = (\ ) \text{ or } \ldots \text{ or } \mathrm{al}_n = (\ ) \\
\qquad \text{then } \mathrm{r} = \mathrm{ival} \\
\qquad \text{else } \mathrm{r} = \mathrm{op}[\mathrm{APPLY}^1[\mathrm{o}_1,\ \text{remove-pseudo}[(\mathrm{a}_{11}\ \mathrm{a}_{21}\ \ldots\ \mathrm{a}_{n1})]], \\
\qquad\qquad \mathrm{FC}[\mathrm{op},\ \mathrm{ival},\ (\mathrm{o}_2\ \mathrm{o}_3\ \ldots\ \mathrm{o}_{t0}), \\
\qquad\qquad\qquad (\mathrm{a}_{12}\ \mathrm{a}_{13}\ \ldots\ \mathrm{a}_{1t1}), \\
\qquad\qquad\qquad (\mathrm{a}_{22}\ \mathrm{a}_{23}\ \ldots\ \mathrm{a}_{2t2}), \\
\qquad\qquad\qquad \quad \vdots \\
\qquad\qquad\qquad (\mathrm{a}_{n2}\ \mathrm{a}_{n3}\ \ldots\ \mathrm{a}_{ntn})]]
\end{array}
$$

-- continued on next page --

1 APPLY is defined as in LISP.

```
remove-pseudo[list] = r
   where list = (sexp1 sexp2 ... sexpn)
      if list = ( )
         then r = ( )
         else if sexp1 = PSEUDO
                 then r = remove-pseudo[(sexp2 sexp3 ... sexpn)]
                 else r = CONS1[sexp1, remove-pseudo[(sexp2 sexp3 ... sexpn)]]
```

[1] CONS is defined as in LISP.

Illustrations

```
?(FC CONS
     NIL
     '(ADD-ELEMENT ADD-ELEMENT)
     '(E1 E1)
     '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC CONS
     NIL
     '(ADD-ELEMENT ADD-ELEMENT)
     '(E1 E1 E1)
     '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC CONS NIL (STAR ADD-ELEMENT) (STAR 'E1) '((E1 E2) (E2 E3 E4)))
((E1 E2) (E1 E2 E3 E4))


?(FC UNION NIL (STAR ADD-ELEMENT) (STAR 'E1) '((E1 E2) (E2 E3 E4)))
(E1 E2 E3 E4)
```

```
?(FC CONS NIL (STAR TIMES) (STAR 5) '(1 2 3))
(5 10 15)


?(FC PLUS 0 (STAR TIMES) (STAR 5) '(1 2 3))
30


?(FC CONS NIL (STAR TIMES PLUS) (STAR 5) '(1 2 3 4 5))
(5 7 15 9 25)


?(FC TIMES 1 (STAR TIMES PLUS) (STAR 5) '(1 2 3 4 5))
118125


?(FC TIMES 1 (STAR TIMES PLUS) '(1 2 3 4 5) (STAR 5))
118125


?(FC CONS NIL (STAR ADD1 PLUS) (STAR 'PSEUDO 5) '(1 2 3 4 5 6))
(2 7 4 9 6 11)


?(FC CONS NIL (STAR TIMES) (STAR 5) NIL)
NIL
```

(FC-APPEND *operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The function FC-APPEND is an EXPR which APPENDs together the results produced by applying *operator-list* to $arg\text{-}list_1,\ldots,arg\text{-}list_n$ as a functional combinator[1]. Given *operator-list*, $arg\text{-}list_1,\ldots,$ and $arg\text{-}list_n$, FC-APPEND uses APPEND to combine the results from the application of *operator-list* as a function combinator to $arg\text{-}list_1,\ldots,$ $arg\text{-}list_n$ with NIL.

Formal Definition

$$\text{FC-APPEND}[\text{ol}, \text{al}_1,\ldots,\text{al}_n] = \text{FC}[\text{APPEND}^2, \text{NIL}, \text{ol}, \text{al}_1,\ldots,\text{al}_n]$$

[1] See FC on page 352 for an explanation of functional combination.

[2] APPEND is defined as in LISP.

<u>Illustrations</u>

```
?(FC-APPEND '(ADD-ELEMENT ADD-ELEMENT)
            '(E1 E1)
            '((E3 E4) (E2 E3 E4)))
(E1 E3 E4 E1 E2 E3 E4)
```

```
?(FC-APPEND '(ADD-ELEMENT ADD-ELEMENT)
            '(E1 E1 E1)
            '((E3 E4) (E2 E3 E4)))
(E1 E3 E4 E1 E2 E3 E4)
```

```
?(FC-APPEND (STAR ADD-ELEMENT) (STAR 'E1) '((E3 E4) (E2 E3 E4)))
(E1 E3 E4 E1 E2 E3 E4)
```

```
?(FC-APPEND (STAR INTERSECT)
            (STAR '(E1 E2))
            '((E1 E3) (E3 E4 E5) (E1 E2)))
(E1 E1 E2)
```

```
?(FC-APPEND (STAR INTERSECT) (STAR '(E1 E2)) NIL)
NIL
```

(FC-CONDCONS *operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The function FC-CONDCONS is an EXPR which returns a list of the non-NIL results produced by applying *operator-list* to $arg\text{-}list_1,\ldots,arg\text{-}list_n$ as a functional combinator[1]. Given *operator-list*, $arg\text{-}list_1,\ldots,$ and $arg\text{-}list_n$, FC-CONDCONS uses CONDCONS[2] to combine the results from the application of *operator-list* as a functional combinator to $arg\text{-}list_1,\ldots,$ $arg\text{-}list_n$ with NIL.

Formal Definition

$$\text{FC-CONDCONS}[\text{ol}, \text{al}_1,\ldots,\text{al}_n] = \text{FC}[\text{CONDCONS}^2, \text{NIL}, \text{ol}, \text{al}_1,\ldots,\text{al}_n]$$

[1]See FC on page 352 for an explanation of functional combination.

[2]CONDCONS is a function of two arguments. If its first argument is NIL, the second argument is returned. Otherwise CONDCONS returns the result of CONSing its first argument onto its second argument.

Illustrations

```
?(FC-CONDCONS '(ADD-ELEMENT ADD-ELEMENT)
              '(E1 E1)
              '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC-CONDCONS '(INTERSECT INTERSECT)
              '((E2) (E2) (E2))
              '((E2 E4) (E3 E4)))
((E2))


?(FC-CONDCONS (STAR INTERSECT)
              (STAR '(E2))
              '((E2 E4) (E3 E4) (E1 E2 E3)))
((E2) (E2))


?(FC-CONDCONS (STAR DIFFERENCE)
              (STAR '(E1 E2 E3))
              '((E1) (E1 E2 E3 E4) (E1)))
((E2 E3) (E2 E3))


?(FC-CONDCONS (STAR DIFFERENCE) (STAR '(E1 E2 E3)) NIL)
NIL
```

(FC-CONS *operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The function FC-CONS is an EXPR which returns a list of the results produced by applying *operator-list* to $arg\text{-}list_1$,..., $arg\text{-}list_n$ as a functional combinator[1]. Given *operator-list*, $arg\text{-}list_1$,..., and $arg\text{-}list_n$, FC-CONS uses CONS to combine the results from the application of *operator-list* as a functional combinator to $arg\text{-}list_1$,...,$arg\text{-}list_n$ with NIL.

Formal Definition

$$\text{FC-CONS}[\text{ol}, \text{al}_1, \ldots, \text{al}_n] = \text{FC}[\text{CONS}^2, \text{NIL}, \text{ol}, \text{al}_1, \ldots, \text{al}_n]$$

[1]See FC on page 352 for an explanation of functional combination.

[2]CONS is defined as in LISP.

Illustrations

```
?(FC-CONS '(ADD-ELEMENT ADD-ELEMENT) '(E1 E1) '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC-CONS '(ADD-ELEMENT ADD-ELEMENT)
          '(E1 E1 E1)
          '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC-CONS (STAR ADD-ELEMENT) (STAR 'E1) '((E3 E4) (E2 E3 E4)))
((E1 E3 E4) (E1 E2 E3 E4))


?(FC-CONS (STAR DIFFERENCE)
          (STAR '(E1 E2 E3))
          '((E3 E4) (E1 E3 E5) (E5) NIL))
((E1 E2) (E2) (E1 E2 E3) (E1 E2 E3))


?(FC-CONS (STAR INTERSECT)
          (STAR '(E1 E2 E3))
          '((E3 E4) (E1 E3 E5) (E5) NIL))
((E3) (E1 E3) NIL NIL)


?(FC-CONS (STAR INTERSECT) (STAR '(E1 E2 E3)) NIL)
NIL
```

(FC-NIL *operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The pseudo-function FC-NIL is an EXPR which applies *operator-list* to $arg\text{-}list_1, \ldots, arg\text{-}list_n$ as a functional combinator[1]. FC-NIL returns NIL.

Formal Definition

$$\text{FC-NIL}[\text{ol}, \text{al}_1, \ldots, \text{al}_n] = \text{FC}[\text{PROGN}^2, \text{NIL}, \text{ol}, \text{al}_1, \ldots, \text{al}_n]$$

[1]See FC on page 352 for an explanation of functional combination.

[2]PROGN is defined as in LISP.

Illustrations

```
?(FC-NIL '(PRINT PRINT) '(E1 E2 E3))
E1
E2
NIL
```

```
?(FC-NIL (STAR PRINT) '(E1 E2 E3))
E1
E2
E3
NIL
```

```
?(FC-NIL (STAR (LAMBDA (A1 A2) (PRINT (LIST A1 A2))))
         (STAR 'ELEMENT)
         '(E1 E2 E3))
(ELEMENT E1)
(ELEMENT E2)
(ELEMENT E3)
NIL
```

```
?(FC-NIL (STAR PRINT) NIL)
NIL
```

(FC-UNION *operator-list* $arg\text{-}list_1$ ... $arg\text{-}list_n$)

Informal Definition

The function FC-UNION is an EXPR which returns the union of the results produced by applying *operator-list* to $arg\text{-}list_1, \ldots,$ $arg\text{-}list_n$ as a functional combinator[1]. Given *operator-list*, $arg\text{-}list_1, \ldots,$ and $arg\text{-}list_n$, FC-UNION uses UNION to combine the results from the application of *operator-list* as a functional combinator to $arg\text{-}list_1, \ldots, arg\text{-}list_n$ with NIL.

Formal Definition

$$\text{FC-UNION}[\text{ol}, \text{al}_1, \ldots, \text{al}_n] = \text{FC}[\text{UNION}, \text{NIL}, \text{ol}, \text{al}_1, \ldots, \text{al}_n]$$

[1]See FC on page 352 for an explanation of functional combination.

Illustrations

```
?(FC-UNION '(INTERSECT INTERSECT)
           '((E1 E2) (E2 E3))
           '((E1 E3) (E1 E3)))
(E1 E3)
```

```
?(FC-UNION '(INTERSECT INTERSECT)
           '((E1 E2) (E2 E3) (E3))
           '((E1 E3) (E1 E3)))
(E1 E3)
```

```
?(FC-UNION (STAR INTERSECT) '((E1 E2) (E2 E3)) (STAR '(E1 E3)))
(E1 E3)
```

```
?(FC-UNION (STAR DIFFERENCE)
           (STAR '(E1 E2))
           '((E1 E3 E5) (E2 E4 E6) (E2)))
(E2 E1)
```

```
?(FC-UNION (STAR DIFFERENCE) NIL '((E1 E3 E5) (E2 E4 E6) (E2)))
NIL
```

(INTERSECT set_1 set_2)[1]

Informal Definition

The function INTERSECT is an EXPR which returns the intersection of set_1 and set_2. Given set_1 and set_2, INTERSECT returns a set consisting of all elements that belong to set_1 and set_2.

Formal Definition

$$
\begin{aligned}
&\mathrm{INTERSECT}[s1,s2] = (e_1 \ldots e_n)\\
&\quad \text{where } s1 = (e_{11} \ldots e_{1k})\\
&\qquad\qquad s2 = (e_{21} \ldots e_{2m})\\
&\qquad\qquad e_i \in \{e_{11},\ldots,e_{1k}\}\\
&\qquad\qquad e_i \in \{e_{21},\ldots,e_{2m}\}
\end{aligned}
$$

[1] "Set" refers to a list with no EQUAL elements.

Illustrations

```
?(INTERSECT '(E1 E2 E4 E5) '(E3 E5 E6))
(E5)
```

```
?(INTERSECT '(E1 E2 E4 E5) '(E3 E6 E7))
NIL
```

```
?(INTERSECT '(E1 E2 E3) '(E1 E2 E3 E4))
(E1 E2 E3)
```

```
?(INTERSECT NIL '(E1 E2 E3 E4))
NIL
```

```
?(INTERSECT '((E1 1) (E 2) (E 3)) '((E 2) (E 3) (E 4)))
((E 2) (E 3))
```

(PRINT-ALL-NEWS *system*)

Informal Definition

The pseudo-function PRINT-ALL-NEWS is an EXPR which prints all the news about *system*. If *system* is GRASPER, PRINT-NEWS prints all editions of the GRASPER news. PRINT-ALL-NEWS returns *system*.

error condition:

- *system* is not GRASPER or LISP

Formal Definition

PRINT-ALL-NEWS[s] = s
 with effects:
  if s = GRASPER
   then where GRASPER-NEWS = (gn_m ... gn_1)
    for each gn_i PRINT[gn_i]
  if s = LISP
   then where LISP-NEWS = (ln_m ... ln_1)
    for each ln_i PRINT[ln_i]

error condition:

- s $\notin$ {GRASPER, LISP}

Illustrations

```
?(PRINT-ALL-NEWS 'GRASPER)

     = = = = = = = = = = = = = = = = = = = = = = = = = =
                         GRASPER NEWS
     = = = = = = = = = = = = = = = = = = = = = = = = = =

                      All editions of the
                         GRASPER News




     -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

GRASPER


?(PRINT-ALL-NEWS 'XXX)
*** PRINT-ALL-NEWS ERROR: NO NEWS FOR XXX
```

(PRINT-NEWS *system*)

Informal Definition

The pseudo-function PRINT-NEWS is an EXPR which prints the most recent edition of the news for *system*. If *system* is GRASPER, PRINT-NEWS prints the most recent edition of the GRASPER news. PRINT-NEWS returns *system*.

error condition:

- *system* is not GRASPER or LISP

Formal Definition

PRINT-NEWS[s] = s

with effects:

if s = GRASPER

then where GRASPER-NEWS = (gn_m ... gn_1)

PRINT[gn_m]

if s = LISP

then where LISP-NEWS = (ln_m ... ln_1)

PRINT[ln_m]

error condition:

- $s \notin$ {GRASPER, LISP}

Illustrations

```
?(PRINT-NEWS 'GRASPER)

        = = = = = = = = = = = = = = = = = = = = = = = = = =
                          GRASPER NEWS
        = = = = = = = = = = = = = = = = = = = = = = = = = =

                    The current edition of the
                          GRASPER News




        ---------------------------------------------------
GRASPER


?(PRINT-NEWS 'XXX)
*** PRINT-NEWS ERROR: NO NEWS FOR XXX
```

(REMOVE-ELEMENT *element set*)[1]

Informal Definition

The function REMOVE-ELEMENT is an EXPR which returns a set resulting from removing *element* from *set*. Given *element* and *set*, REMOVE-ELEMENT returns *set* if *element* is not contained in *set*. If *element* is in *set*, REMOVE-ELEMENT returns a set containing all the elements in *set* except *element*.

Formal Definition

REMOVE-ELEMENT[e,s] = DIFFERENCE[s,(e)]

[1] "Set" refers to a list with no EQUAL elements.

Illustrations

```
?(REMOVE-ELEMENT 'E2 '(E1 E2 E3))
(E1 E3)

?(REMOVE-ELEMENT 'E2 '(E1 E3 E4))
(E1 E3 E4)

?(REMOVE-ELEMENT 'E2 NIL)
NIL

?(REMOVE-ELEMENT '(E 2) '((E 1) (E 2) (E 3)))
((E 1) (E 3))
```

(STAR *s-expression*$_1$ ... *s-expression*$_n$)[1]

Informal Definition

The function STAR is an EXPR which returns a list where the sequence *s-expression*$_1$,...,*s-expression*$_n$ is repeated an infinite number of times. Given *s-expression*$_1$,...,*s-expression*$_n$, STAR returns an infinite list consisting of *s-expression*$_1$,..., *s-expression*$_n$ repeated an infinite number of times[2].

Formal Definition[3]

$$\text{STAR}[s_1,s_2,\ldots,s_n] = (s_1\ s_2\ \ldots\ s_n\ s_1\ s_2\ \ldots\ s_n\ \ldots)$$

[1]STAR is primarily intended to be used in conjunction with the functional combinator operators defined on pages 356-364.

[2]Any attempt to print the result of this function will result in an infinite loop.

[3]This can be implemented by simulating the infinite result with a circular list (i.e., a list whose final CDR points back to the head of the list).

Illustrations

```
?(CAR (STAR 'E))
E

?(CADR (STAR 'E))
E

?(CADDR (STAR 'E))
E

?(CAR (STAR 'E1 'E2))
E1

?(CADR (STAR 'E1 'E2))
E2

?(CADDR (STAR 'E1 'E2))
E1
```

(UNION set_1 set_2)[1]

Informal Definition

The function UNION is an EXPR which returns the union of set_1 and set_2. Given set_1 and set_2, UNION returns a set consisting of all elements that belong to set_1 or to set_2.

Formal Definition

$$
\begin{aligned}
&\mathrm{UNION}[s1,s2] = (e_1 \ldots e_n) \\
&\quad \text{where } s1 = (e_{11} \ldots e_{1k}) \\
&\qquad\quad s2 = (e_{21} \ldots e_{2m}) \\
&\qquad\quad e_i \in \{e_{11},\ldots,e_{1k}\} \text{ or } e_i \in \{e_{21},\ldots,e_{2m}\} \\
&\qquad\quad \{e_{11},\ldots,e_{1k}\} \subseteq \{e_1,\ldots,e_n\} \\
&\qquad\quad \{e_{21},\ldots,e_{2m}\} \subseteq \{e_1,\ldots,e_n\}
\end{aligned}
$$

[1]"Set" refers to a list with no EQUAL elements.

Illustrations

```
?(UNION '(E1 E2 E4 E5) '(E3 E5 E6))
(E1 E2 E4 E3 E5 E6)


?(UNION '(E1 E2 E4 E5) '(E3 E6 E7))
(E1 E2 E4 E5 E3 E6 E7)


?(UNION '(E1 E2 E3) '(E1 E2 E3 E4))
(E1 E2 E3 E4)


?(UNION '(E1 E2) '(E1 E2))
(E1 E2)


?(UNION NIL '(E1 E2))
(E1 E2)


?(UNION '((E 1) (E 2) (E 4) (E 5)) '((E 3) (E 5) (E 6)))
((E 1) (E 2) (E 4) (E 3) (E 5) (E 6))
```

Appendix D

Error Messages

This appendix contains all of the messages printed in response to GRASPER errors. This appendix does not include the messages printed in response to LISP errors. The italicized portions of each error message are variables whose values are printed. The name of each of these variables indicates the type of entity bound to the variable. The error messages are in alphabetical order (disregarding variables). Each error message is followed by a brief description of the condition causing the error and a list of the GRASPER operators that can generate that condition.

---

grasper-operator ERROR: A SPACE NAMED *non-space* CANNOT BE CREATED SINCE THAT NAME SIGNIFIES A VIRTUAL SPACE

This message is printed when an attempt is made to create a space with a name that signifies a virtual space.

grasper-operator = CUS or CREATE-GRAPH

---

grasper-operator ERROR: *grasper-operation* CANNOT REFERENCE THE UNIVERSAL SPACE WHEN IT IS VIRTUAL

This message is printed when an operator which is only defined over real spaces references UNIVERSE while it is virtual.

grasper-operator = BAP, BIP, BOP, BUN, BUS, CAP, CIP, COP, CREATE-GRAPH, CUN, DAG, DAGG, DAGN, DAN, DANG, DAP, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, DUN, VAP, VIP, VOP, VUN, or VUS

---

grasper-operator ERROR: FILE *file* IS NOT A GRASPER FILE

This message is printed when INPUT-GRAPH is called with a file that does not contain a GRASPER-GRAPH.

grasper-operator = INPUT-GRAPH

---

grasper-operator ERROR: *illegal-size-parameter* IS NOT A LEGAL SIZE PARAMETER
THE LEGAL SIZE PARAMETERS ARE:
1. *legal-size-parameter*$_1$
2. *legal-size-parameter*$_2$
⋮

This message is printed when a GRASPER operator is called with an unknown size parameter.

grasper-operator = SET-SIZE or SIZE

---

grasper-operator ERROR: *non-node* IS NOT A NODE

This message is printed when a non-existent node is referenced.

grasper-operator = BAP, BIP, BOP, BUN, CAP, CIP, COP, CREATE-GRAPH, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, PRINT-NODE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, SUS, VAP, VIP, VOP, VUN, XAP, XIP, or XOP

---

grasper-operator ERROR: *non-node* IS NOT A NODE IN SPACE *space*

This message is printed when a reference is made to a node in a space (other than UNIVERSE) that does not exist in that space.

grasper-operator = BAP, BIP, BOP, BUN, CAP, CIP, COP, CREATE-GRAPH, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, PRINT-NODE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, VAP, VIP, VOP, VUN, XAP, XIP, or XOP

---

grasper-operator ERROR: *non-space* IS NOT A SPACE

This message is printed when a non-existent space is referenced.

grasper-operator = BAP, BIP, BOP, BUN, BUS, CAP, CIP, COP, CREATE-GRAPH, CUN, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DESCRIBE-SPACE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, DUN, PRINT-NODE, PRINT-SPACE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, SUN, VAP, VIP, VOP, VUN, VUS, XAP, XIP, XOP, or XUN

---

grasper-operator ERROR: NO NEWS FOR *non-news-system*

This message is printed when no news exists for the referenced system.

grasper-operator = PRINT-ALL-NEWS or PRINT-NEWS

---

grasper-operator ERROR: NO SIZE PARAMETER WAS PROVIDED
THE LEGAL SIZE PARAMETERS ARE:
1. *legal-size-parameter*$_1$
2. *legal-size-parameter*$_2$
⋮ ⋮

This message is printed when a GRASPER operator requiring at least one size parameter is called with no arguments.

grasper-operator = SET-SIZE or SIZE

---

grasper-operator ERROR: PAGE SIZE CANNOT BE LARGER THAN MEMORY SIZE
THE CONFLICTING VALUES WERE:
MEMORY SIZE = *memory-size*
PAGE SIZE = *page-size*

This message is printed when an attempt is made to set the maximum page size larger than the maximum memory size.

grasper-operator = SET-SIZE or INPUT-GRAPH

---

grasper-operator ERROR: POORLY FORMED GRAPH-DESCRIPTOR
BAD SPACE-VALUE-DESCRIPTOR
THE SPACE-VALUE-DESCRIPTOR WAS *non-space-value-descriptor*

This message is printed when a non-NIL atom is used as a space-value-descriptor.

grasper-operator = CREATE-GRAPH

---

grasper-operator ERROR: POORLY FORMED GRAPH-DESCRIPTOR
THE GRAPH-DESCRIPTOR WAS *non-graph-descriptor*

This message is printed when a non-NIL atom is used as a GRAPH-descriptor.

grasper-operator = CREATE-GRAPH

---

grasper-operator ERROR: POORLY FORMED NODE DESCRIPTOR
THE NODE-DESCRIPTOR WAS *node-descriptor*

This message is printed when an atom or a list with more than four elements is used as a node-descriptor.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: POORLY FORMED NODE-DESCRIPTOR
BAD LIST OF INPOINTING-PAIR-DESCRIPTORS ASSOCIATED WITH NODE *node*
THE INPOINTING-PAIR-DESCRIPTOR LIST WAS *non-inpointing-pair-descriptor-list*

This message is printed when a non-NIL atom is used as an inpointing-pair-descriptor list.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: POORLY FORMED NODE-DESCRIPTOR
BAD LIST OF OUTPOINTING-PAIR-DESCRIPTORS ASSOCIATED WITH NODE *node*
THE OUTPOINTING-PAIR-DESCRIPTOR LIST WAS *non-outpointing-pair-descriptor-list*

This message is printed when a non-NIL atom is used as an outpointing-pair-descriptor list.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: POORLY FORMED NODE-DESCRIPTOR
BAD INPOINTING-PAIR-DESCRIPTOR ASSOCIATED WITH NODE *node*
THE INPOINTING-PAIR-DESCRIPTOR WAS *non-inpointing-pair-descriptor*

This message is printed when an S-expression used as an inpointing-pair-descriptor is not a list of the appropriate length or it contains a non-NIL atom used as an inpointing-pair-space-value-descriptor.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: POORLY FORMED NODE-DESCRIPTORS
BAD NODE-SPACE-VALUE-DESCRIPTOR ASSOCIATED WITH NODE *node*
THE NODE-SPACE-VALUE-DESCRIPTOR WAS *non-node-space-value-descriptor*

This message is printed when a non-NIL atom is used as a node-space-value-descriptor.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: POORLY FORMED NODE-DESCRIPTOR
BAD OUTPOINTING-PAIR-DESCRIPTOR ASSOCIATED WITH NODE *node*
THE OUTPOINTING-PAIR-DESCRIPTOR WAS *non-outpointing-pair-descriptor*

This message is printed when an S-expression used as an outpointing-pair-descriptor is not a list of the appropriate length or it contains a non-NIL atom used as an outpointing-pair-space-value-descriptor.

grasper-operator = CREATE-GRAPH or CREATE-NODE

---

grasper-operator ERROR: *grasper-operator* REQUIRES AN EVEN NUMBER OF ARGUMENTS
THE ARGUMENTS PROVIDED WERE:

1. $argument_1$
2. $argument_2$

$\vdots \quad \vdots$

This message is printed when a GRASPER operator requiring an even number of arguments is called with an odd number of arguments.

grasper-operator = SET-SIZE

---

grasper-operator ERROR: *size-parameter* SIZE MUST BE A POSITIVE INTEGER
THE SIZE PROVIDED WAS *bad-size*

This message is printed when an attempt is made to set a GRASPER size parameter to something other than a positive integer.

grasper-operator = SET-SIZE

---

grasper-operator ERROR: THE OUTPOINTING AND INPOINTING EDGES *edge* BETWEEN NODE $node_1$ AND NODE $node_2$ DO NOT HAVE EQUAL VALUES

This message is printed when the outpointing and inpointing edges referenced by VAP do not have the same value (in UNIVERSE).

grasper-operator = VAP

---

grasper-operator ERROR: THE OUTPOINTING AND INPOINTING EDGES *edge* BETWEEN NODE $node_1$ AND NODE $node_2$ DO NOT HAVE EQUAL VALUES IN SPACE *space*

This message is printed when the outpointing and inpointing edges referenced by VAP do not have the same value in the referenced space (other than UNIVERSE).

grasper-operator = VAP

---

grasper-operator ERROR: THE SECOND ARGUMENT MUST BE A LIST OF SPACES
THE SECOND ARGUMENT PROVIDED WAS *non-list-of-spaces*

This message is printed when a non-NIL atom is used as the second argument instead of a list of spaces.

grasper-operator = DESCRIBE-NODE or PRINT-NODE

---

grasper-operator ERROR: THERE IS NO EDGE *non-edge* BETWEEN NODE $node_1$ AND NODE $node_2$

This message is printed when both the outpointing and inpointing referenced edges do not exist.

grasper-operator = BAP or VAP

---

grasper-operator ERROR: THERE IS NO EDGE *non-edge* BETWEEN NODE $node_1$ AND NODE $node_2$ IN SPACE *space*

This message is printed when both the outpointing and inpointing referenced edges do not exist in the referenced space (other than UNIVERSE).

grasper-operator = BAP or VAP

---

grasper-operator ERROR: THERE IS NO EDGE *non-edge* POINTING FROM NODE $node_1$ TO NODE $node_2$

This message is printed when a non-existent edge is referenced.

grasper-operator = BIP, BOP, VIP, or VOP

---

grasper-operator ERROR: THERE IS NO EDGE *non-edge* POINTING FROM NODE $node_1$ TO NODE $node_2$ IN SPACE *space*

This message is printed when a reference is made to an edge in a space (other than UNIVERSE) that does not exist in that space.

grasper-operator = BIP, BOP, VIP, or VOP

---

grasper-operator ERROR: TOO FEW ARGUMENTS WERE PROVIDED
grasper-operation REQUIRES AT LEAST *min* ARGUMENT(s)
NONE WERE PROVIDED

This message is printed when a GRASPER operator (with an optional last argument) is called with no arguments and requires at least one.

grasper-operator = BAP, BIP, BOP, BUN, CAP, CIP, COP, CUN, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, DUN, PRINT-NODE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, VAP, VIP, VOP, VUN, XAP, XIP, XOP, or XUN

---

grasper-operator ERROR: TOO FEW ARGUMENTS WERE PROVIDED
grasper-operation REQUIRES AT LEAST *min* ARGUMENT(s)
THE ARGUMENT(s) PROVIDED WERE:
1. $argument_1$
2. $argument_2$
$\vdots \quad \vdots$

This message is printed when a GRASPER operator (with an optional last argument) is called with too few arguments.

grasper-operator = BAP, BIP, BOP, BUN, CAP, CIP, COP, CUN, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, DUN, PRINT-NODE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, VAP, VIP, VOP, VUN, XAP, XIP, XOP, or XUN

---

grasper-operator ERROR: TOO MANY ARGUMENTS WERE PROVIDED

grasper-operator CAN HAVE NO MORE THAN *max* ARGUMENT(s)

THE ARGUMENTS PROVIDED WERE:

1. $argument_1$
2. $argument_2$

$\vdots \quad \vdots$

This message is printed when a GRASPER operator (with an optional last argument) is called with too many arguments.

grasper-operator = BAP, BIP, BOP, BUN, CAP, CIP, COP, CUN, DAG, DAGG, DAGN, DAN, DANG, DAP, DESCRIBE-NODE, DIG, DIGG, DIGN, DIN, DING, DIP, DOG, DOGG, DOGN, DON, DONG, DOP, DUN, PRINT-NODE, SAG, SAGN, SAN, SANG, SAP, SIG, SIGN, SIN, SING, SIP, SOG, SOGN, SON, SONG, SOP, SUN, SUS, VAP, VIP, VOP, VUN, XAP, XIP, XOP, or XUN

BIBLIOGRAPHY

BIBLIOGRAPHY

[ALL78] John R. Allen (1978). The Anatomy of LISP. McGraw-Hill Book Co., New York, New York.

[FRI74] Daniel P. Friedman (1974). The Little LISPer. SRA, Palo Alto, California.

[FRI77] Daniel P. Friedman and David S. Wise (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.

[KON75] Kurt Konolige (1975). ALISP User's Manual. University of Massachusetts Computing Center, Graduate Research Center, 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 M.I.T. Press, Cambridge, Massachusetts.

[NOR78a] Mat Nordström (1978). LISP F3: Implementation Guide and System Description. Datalogilaboratoriet, Sturegatan 2B, S-752 23 Uppsala, Sweden.

[NOR78b] Mat Nordström (1978). LISP F3: Users Guide. Datalogilaboratoriet, Sturegatan 2B, S-752 23 Uppsala, Sweden.

[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.

[SIK76] Laurent Siklossy (1976). Let's Talk LISP. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

[WEB70] Webster's Seventh New Collegiate Dictionary (1977). G. & C. Merriam Co., Springfield, Massachusetts.

[WIN77] Patrick H. Winston (1977). Artificial Intelligence. Addison-Wesley Publishing Company, Reading, Massachusetts.

[WIS67] C. Weissman (1967). LISP 1.5 Primer. Dickenson Publishing Co., Belmont, California.

INDEX/GLOSSARY

INDEX/GLOSSARY

Carteisan product of set_1 with set_2, i.e., $\{(e_1\ e_2) | e_1 \in set_1, e_2 \in set_2\}$.

⊕: "*string/sequence*$_1$ ⊕ *string/sequence*$_2$" represents the concatenation of *string/sequence*$_1$ with *string/sequence*$_2$, 265.

?: The prompt character in all illustrations.

': Macro character for QUOTE, a LISP function of one argument that returns its unevaluated argument as its value, e.g., the result of evaluating 'ARGUMENT is ARGUMENT.

|: "{*element*|*condition*}" represents a set containing all choices of *element* that satisfy *condition*.

: "$expression_0 = expression_1 | expression_2 | \ldots | expression_n$" is an abbreviated form of ($expression_0 = expression_1$) or ($expression_0 = expression_2$) or ... or ($expression_0 = expression_n$).

:= : "*variable* := *expression*" indicates that *variable* is assigned the value of *expression*.

< >: Sequence delineators, 265.

{ }: Set delineators.

[ ]: Argument list delineators; "*operator*[$argument_1 \ldots argument_n$] represents an application of *operator* with $argument_1 \ldots argument_n$ as arguments.

( ): List delineators.

: Operator precedence indicators.

∃: Abbreviation for "there exists".