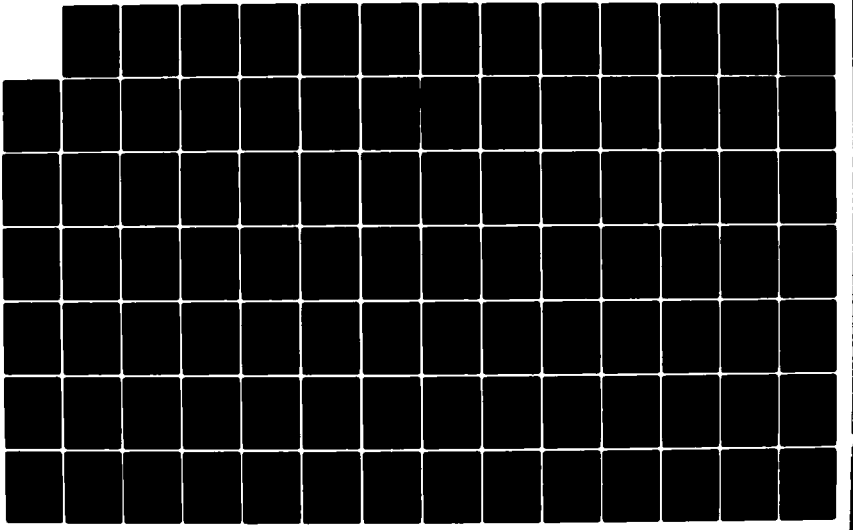
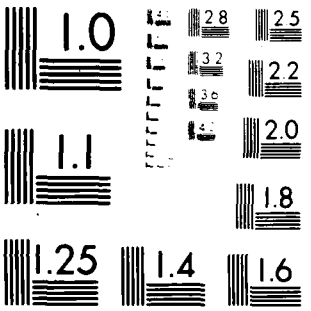
AD-A122 360 GRAPES USER'S MANUAL(U) CARNEGIE-MELLON UNIV PITTSBURGH 1/2
PA DEPT OF PSYCHOLOGY R SAUERS ET AL. NOV 82 ONR-82-3
N00014-81-C-0335

UNCLASSIFIED F/G 9/2 NL





GRAPES

User's Manual

Ron Sauers
Dept. of Mathematics
Carnegie-Mellon University

Robert Farrell
Department of Psychology
Carnegie-Mellon University

November 1982

This research was supported by the Personnel and Training Research Programs, Psychological Services Division, Office of Naval Research, under Contract No.: N00014-81-C-0335, Contract Authority Identification Number, NR No.: 157-465 to John Anderson.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: ONR-82-3	2. GOVT ACCESSION NO.: AD-A122 360
	3. RECIPIENT'S CATALOG NUMBER:
4. TITLE (and Subtitle): GRAPES User's Manual	5. TYPE OF REPORT & PERIOD COVERED: Interim report
	6. PERFORMING ORG. REPORT NUMBER:
7. AUTHOR(s): Ron Sauers Robert Farrell	8. CONTRACT OR GRANT NUMBER(s): N00014-81-C-0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Psychology Carnegie-Mellon University Pittsburgh, PA 15213	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: NR 157-465
11. CONTROLLING OFFICE NAME AND ADDRESS: Personnel and Training Research Programs Office of Naval Research Arlington, VA 22217	12. REPORT DATE: November 30, 1982
	13. NUMBER OF PAGES: 101
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report)
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

production system	composition	planning	compilation
programming language	pattern matching	automatic programming	data-flow
LISP	working memory	cognitive	
goal structure	interpreter	computer	
proceduralization	modelling	learning	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

GRAPES is a goal-restricted production system designed for modeling human cognitive processes. The system's declarative knowledge resides in a dynamic working memory and its procedural knowledge is embodied in condition-action rules known as productions. Each production can apply only in reference to the current goal. The goals are organized in an and/or tree with the root node being the top goal, which is specified by the user. The tree is explored in a left-to-right, depth-first manner. Features of the language

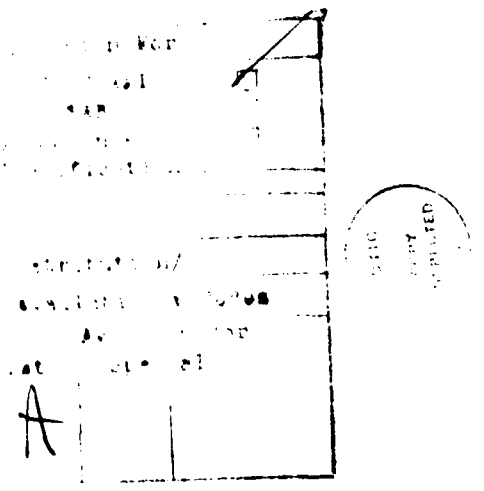
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. Abstract (cont)

include goal parameter specification, flexible goal matching, LISP function calls within production rules, and a host of user-acessible functions designed for their powerful matching ability. The interpreter includes an interrupt capability, a photo package, a tracing mechanism, and various debugging facilities. One peripheral module contains procedural-ization and composition, two learning mechanisms used to acquire new productions. Another module contains useful functions for modelling LISP programmers. GRAPES is best-suited for highly goal-directed tasks involving planning or problem solving.

Table of Contents



1. Introduction

GRAPES is an interpreter for a Goal-Restricted Production System language. This document is intended to allow the user to create and use GRAPES program environments in order to accomplish goal-directed tasks. The GRAPES interpreter is built on top of a Franz LISP environment and therefore a basic familiarity with some dialect of LISP is assumed. Previous knowledge of a production system language (such as OPS5) is helpful, but not essential.

Users of the GRAPES interpreter do not really write programs; instead, they must create program environments. A GRAPES program environment consists of three major pieces. These are:

Working Memory This part of the program environment holds what, in most high-level programming languages, would be considered program data.

A Goal Structure This provides a great deal of the control structure within the program environment, and is basically a means of letting the system know what tasks it is supposed to accomplish.

A Production Set This part of the environment is the part which most people would recognize as doing most of the work while the system is running. Each production in the production set is a *miniature program* which can test against and operate on working memory and the goal structure while the GRAPES system is running.

Although the three pieces of the GRAPES program environment are dependent on each other, each is a separate entity. For this reason, each of the pieces will be described in its own section within this document. Section two describes the working memory structure, section three describes the goal structure, and section four describes the GRAPES production set. Section five describes the recognize-act cycle, including GRAPES's conflict resolution strategies. Then, in part six, a section is devoted to a special set of functions designed to allow the user to write GRAPES productions more easily. Section seven describes how to use the GRAPES interpreter in order to execute a program environment. Section eight is devoted to the GRAPES debugging mechanisms. In section nine three special i/o packages are introduced. Finally, in section ten, the GRAPES learning mechanisms are explained. A sample run and a glossary of commands can be found in the appendices, as well as a syntactic specification of GRAPES productions.

2. Working Memory

Working memory is that part of the GRAPES program environment which stores data elements. The user may initialize working memory so that it contains some data at the start of system execution[1] . Also, data items may be inserted or deleted during system execution.

Each data item in working memory is called a working memory element. Each working memory element is a list structure[2] , and can have an arbitrary length and complexity. For example, the following structure is a legal GRAPES working memory element:

```
(IF list1 HAS-RELATIONSHIP relation1 TO (list2)
    THEN either (list1 HAS-RELATIONSHIP relation2 TO (list2))
             OR ((list1 HAS-RELATIONSHIP relation2 TO (list3))
                 AND
                 (list3 HAS-RELATIONSHIP relation1 TO (list2))))
```

However, working memory elements can be much simpler. For example, this is also a legal working memory element:

```
(IS-A argument1 ATOM)
```

The following are **not** legal working memory elements:

```
working-memory-element                  ; Not a list.
(IS-A (item1) a (list)))                ; Unmatched parenthesis.
```

Thus, it can be seen that fairly complicated data structures can be represented in working memory. The most important constraint is that working memory elements should contain information which is relevant to the task at hand. Also, the user may want to adopt some kind of convention in forming working memory elements, since this helps in writing production sets which work properly. For example, some conventions which might be used are:

```
(HAS-RELATIONSHIP <item1> <relationship> ( <list-of-items> ))
```

or

```
(IS-A <item> <type> )
```

---

[1] see section seven

[2] in the sense of a LISP list structure

3. The Goal Structure

The GRAPES goal structure is that part of the program environment which tells the system what tasks it is supposed to be accomplishing at any given time. The user must start the system off with an initial goal, called the *top-goal*. The main objective of the system is to succeed with the top-goal; once the top-goal is successful, the system stops running.

As the system runs, more goals are created and inserted into the goal tree. For example, a goal (let us call it goal-1) may get inserted as a subgoal of the top-goal, and then two other goals (goal-2 and goal-3) may become subgoals of goal-1, etc. As each new goal is inserted into the goal tree, the system tries to accomplish that goal. This process continues until the system finds a goal which it can solve; this goal becomes successful. A goal is successful when a production explicitly declares it to be successful[3], or when all of its subgoals have been successful.

At the beginning of each cycle, GRAPES does a depth-first search on the goal tree, looking for a goal which has not already been successful. If there is no such goal[4], then the system stops, having accomplished its top-goal. Otherwise, the resulting goal is declared to be the *current-goal*. When the system first begins to run, the top-goal automatically becomes the current-goal.

Thus, *at all times, the system is operating in the context of a current-goal*. This goal is important, since all productions must *fire* in the context of the current-goal. The user will find that this gives the system a well-defined control structure, while still retaining the flexibility of other production system languages.

3.1. Goal Format

All goals are specified by a set of parameters,each of which is an attribute/value pair. Each attribute must be an atom ending in the character ":". Values may be any legal symbolic expression. There is no restriction on the number of parameters which a goal may have, or on the names of the attributes, except that every legal goal must have an "action:" attribute.

For example, the following is a legal format for describing the top-goal:

```
(top-goal
        (action: write-function
         name: function1
         arguments: (arg1 arg2 arg3)
         output: result1) )
```

---

[3] see section six

[4] that is, if every goal in the tree has been successful

In this example, the "action:" of the top-goal is "write-function". The other attributes defined for the top-goal are "name:", "arguments:", and "output:". Note that values given to attributes can be either atoms or lists.

The following examples are **not** legal descriptions for a goal:

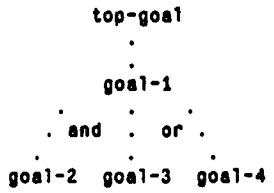
```
(top-goal
        (object: goal-1
         argument: arg1) )      ; This goal has no action:.

(top-goal
        (action: some-action    ; "argument" is not a legal
         argument arg1) )       ; attribute- there is no ":".
```

3.2. Creating an And-Or Tree

The goal tree that the system constructs is, by default, an *and* tree. Each branch must be executed in a left to right manner. An *or* branch can be achieved by firing a production at a goal which failed, or through the use of the *redo command. If either of these techniques is used, the given goal will be an *or* branch, and the state of the goal tree will be preserved while the new goal is inserted.

For example, if the goal tree is:

and we did a (*redo goal-1) then goal-1 would become an active goal. If a production fired at goal-1 and inserted a new subgoal, goal-4, then the goal tree would look like this:



where goal-4 is now an or branch and will be the only subgoal recognized be the interpreter. Future implementations make make use of the old goal information[5]. Or branches can also be made when a goal fails and another production fires at the goal. At each failure, the system will try to find any applicable

[5] attached to goal-2 and goal-3 in the example

productions, but if no such productions are found, the failure will continue up the tree to the top goal.

Productions can match against a goal or a block of goals, anywhere in the goal tree. In addition, goals can be inserted at any place in the tree, one at a time or as a whole unit. This gives the flexibility needed for modelling highly demanding cognitive tasks, where subjects do not necessarily follow a strict depth-first search strategy. See section 4 for a more detailed description of goal matching.

4. The Production Set

The GRAPES production set is the part of the program environment which tells the system when and how the rest of the program environment may change. Each production in the production set is in itself a miniature program description.

4.1. Production Format

Each production has two major portions. The first piece is called the *left-hand side* of the production, and is basically a description of the conditions which must be true in the GRAPES environment in order for that production to *fire*. The other piece, called the *right-hand side* of the production, is a description of what actions are to be taken when that production fires.

Productions are defined using the GRAPES function *p*. The two halves of the production are separated by the symbol ==>. Also, each production must be given a unique name, which must be a non-nil atom having no Franz LISP function definition. Thus, the syntax for a production definition is:

```
(p <production-name>
      <left-hand-side>
   ==>
      <right-hand-side> ).
```

Or more specifically:

```
(p <production-name>
   action: <value>
   [ <other parameters> ]
   [ <tests:>  [ <working-memory tests> ]
               [ <goal tests> ]
               [ <function calls> ]
              ...
              ...                        ]
==>
  [ <working memory additions> ]
  [ <goal tree additions > ]
  [ <function calls> ]
  ...
  ...                              )
```

The remainder of this section describes in detail the formats of each of the sections of a production. For a precise syntactic specification for GRAPES productions, see appendix eight.

4.2. The Left-Hand Side

The left-hand side is that part of a production which describes the conditions which must be true in the GRAPES program environment in order for that production to *fire*. The left-hand side of a production always specifies a context which the system must be operating under[6]. It may also test against the data in working memory, and may test against the contents of the goal tree.

4.2.1. Pattern Matching

The tests on the left-hand side are accomplished through a process known as *pattern matching*. This means that each test is a *pattern* (a parameterized description of what something must "look like") which must *match* against some data item in working memory or in the goal tree[7].

Since patterns are of major importance when writing GRAPES productions, they will be described first. Then, a few sections will be devoted to explaining how these patterns are used to form tests on the left-hand side of a production.

4.2.1.1. Constant Patterns

The simplest type of pattern is a single constant. Constants specify exactly the data item which is to be matched. Any *ordinary* atom (that is, an atom which is not a variable) is by default defined to be a constant. For example,

```
has-relationship
```

is a legal GRAPES pattern which only matches against the data item:

```
has-relationship
```

Patterns can be constructed by forming a list from other patterns. For example, the following are also legal GRAPES patterns:

```
(is-a LISP programming-language)
(LISP has-functions named (car cdr cons))
```

---

[6] the left-hand side must contain a specification of the current-goal

[7] these items must fit the given description

4.2.1.2. Variable Patterns

Patterns would not be very useful if they could only contain constants. This, however, is not the case. Patterns are permitted to contain variables. A variable (actually, an ordinary variable) is an atom which begins with the character =. Variables do not specify exactly how the matched data item looks; instead, they enable the user to describe a class of data items which will match to a pattern. Variables can match to any legal symbolic expression. For example, the following is a legal GRAPES pattern:

```
=something
```

which will match to any of the following data items:

```
some-constant
(a list)
(a more (complex) ((list-structure)))
```

Variables can be part of a larger pattern. For example, the pattern

```
(has-relation =argument1 =relation =argument2)
```

will match to any of the following data items:

```
(has-relation list1 first (list1 list2 list3))
(has-relation list1 same-as list1)
(has-relation green (some physical-property) (grass crayons etc)).
```

4.2.1.3. Variable Bindings

Variables have a very important property. During the match process they become *bound* to the data item or portion of a data item which they match to. The value which a variable becomes bound to is called its *binding*. For example, when the pattern

```
(has-relation =argument1 =relation =argument2)
```

is matched against the data item

```
(has-relation green (some physical-property) (grass crayons etc))
```

the variable *=argument1* becomes bound to the atom *green*, the variable *=relation* becomes bound to the list *(some physical-property)*, and the variable *=argument2* becomes bound to *(grass crayons etc)*.

When the same variable occurs more than once in the same pattern (or, as we shall see later, in the same production), it must bind to the same data item each time. Thus, the pattern

```
(has-relation =arg1 same-as =arg1)
```

will match to the data item

```
(has-relation list1 same-as list1)
```

but not to any of these data items:

```
(has-relation list1 same-as list2)
(has-relation list1 same-as LIST1)
(has-relation list1 same-as (list1))
```

4.2.1.4. String Variables

Patterns may also contain another type of variable, called a *string variable* (or a *segment variable*). A string variable is an atom which begins with the character "$". String variables are similar to regular variables, except that they bind to a series of (zero or more) expressions within a data item. String variables always bind to a list (or nil), but this list appears *spliced* within the data item (that is, each element in the binding of a string variable is at the same depth of list structure as the surrounding terms in the data item).

For example, when the pattern

```
(has-relation arg1 first (arg1 $remaining-items))
```

is matched against the data item

```
(has-relation arg1 first (arg1 arg2 arg3))
```

the string variable "$remaining-items" becomes bound to the list "(arg2 arg3)".

String variables are meant to be used to matched pieces of a list. They will only match zero items when the list is non-empty. Matching against nil can be done with calls to LISP (explained in section 4.1.5).

Note that sometimes string variables may introduce ambiguity into a pattern. However, this is often desirable. For example, if the pattern

```
($list1 =term $list2)
```

is matched against the data item

```
(arg1 arg2 arg3),
```

then there are three possible sets of variable bindings which result:

1. $list1 bound-to nil
   =term bound-to arg1
   $list2 bound-to (arg2 arg3)

2. $list1 bound-to (arg1)
   =term bound-to arg2
   $list2 bound-to (arg3)

3. $list1 bound-to (arg1 arg2)
   =term bound-to arg3
   $list2 bound-to nil

4.2.1.5. Patterns Returned from Functions

There is one remaining type of entity which may occur within a GRAPES pattern. This is an external function call. External function calls will be described in detail in section five. However, as an example if the top-goal has no subgoals, then the pattern

```
(is-a (*subgoals top-goal) empty-list)
```

will match to the data item

```
(is-a nil empty-list)
```

4.2.2. Production Parameters

Production parameters specify the context which the system is operating under. Each production fires in the context of the current goal. Production parameters are attribute/value pairs, entirely analogous to those of the goals in the goal tree, except that the values can be GRAPES patterns (as described in section 4.1.1). Each production must have an *action:* parameter, whose value is a non-nil atom. All other parameters are optional. For a parameter match to be successful, the current goal must have that parameter's attribute and the corresponding value must match the specified pattern. However, the current goal can have other attributes not specified in the parameter list, and the parameter patterns will still match. For example, if

```
(p p-1
  action: make
  object: =thing
  tests:
      (is-a =thing good)   .
==>
  (make =thing) )
```

was a production in production memory, and

```
(goal-4
      (action: make
       object: cake
       color: brown))
```

was the current goal, then p-1 would parameter match to goal-4. After parameter matching would come working memory matching and goal matching. If all of these tests passed, then the production would be placed in the *conflict set*[8] .

---

[8] *see section five*

4.2.3. Working Memory Tests

All GRAPES working memory tests are in the *tests:* section of the production. This section is an optional part of a GRAPES production. Often the goal context alone is a sufficient precondition for the production to fire. GRAPES working memory tests are simply patterns, as described in section 4.1.1. These patterns are matched against working memory. The only restriction on working memory tests is that they may not begin with the constants *goal*, *subgoal*, or *supergoal*, and they may not make references to external functions which are not defined (however, the user can always define his own external functions).

These are examples of legal working memory tests:

```
(has-relation =var =relation ($argset))
(has-color (*pattern =dog (*length $colorset)) =color)
```

These are examples of illegal working memory tests:

```
(goal pick-up =block)                ; Begins with a goal-word
(has-relation (*make-pattern =this)) ; Undefined function*make-pattern
```

4.2.4. Goal Tests

Like working memory tests, GRAPES goal tests are in the *tests:* section of the production. GRAPES maintains the goal tree internally. Many features have been added to make goal matching as flexible as possible. A goal specification is a pattern of the form:

```
( <type> [ <of-goal> [ <name> ]]
           <parameters> [ <goal1> [ <goal2> ... ]] )
```

<type> Either *goal*, *subgoal*, or *supergoal*.

<of-goal> A goal name (not valid for type *goal*), which specifies that the search for subgoal or supergoal begins here.

<name> The name of the goal with the given parameters.

<parameters> The list of goal parameters. For example:

```
(action: write
 function: =name)
```

<goal n> if given, each is a nested goal specification.

Specifying a type *goal* in the goal test means that any active goal may try to match to this specification Specifying the type *supergoal* means that any active goal which is higher in the goal tree than <of-goal> may be considered for matching. Specifying the type *subgoal* means that any active goal which is lower in the goal

tree than <of-goal> may be considered for matching.

4.2.4.1. Nested Goal Tests

The goal name <of-goal> defaults to the current goal in an outer level goal test. If a goal test is being nested, the following rules specifiey the the default value of <of-goal>:

1. Type *goal* - an immediate subgoal of the enclosing goal specification.

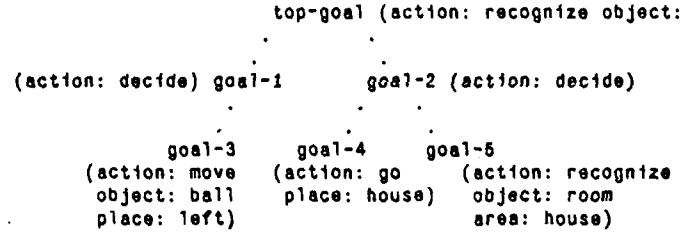
2. Type *subgoal* or *supergoal* - the goal matched by the enclosing goal specification.

Note that if any nested goal test fails, then the entire test fails.

4.2.4.2. Goal Test Parameters

The parameters are analogous to those described in section 3. They consist of attribute/value pairs. Each attribute must be an atom, but the values may be arbitrary GRAPES expressions (however, they must evaluate to a non-nil atom or list). Parameters are optional.

If the goal tree were as follows:



and the current goal was goal-2, then the following would be legal goal tests:

TESTS:	MATCHES:
(goal =goal (action: recognize object: =object))	top-goal goal-6
(supergoal (action: recognize object: (=bind =name ball)))	top-goal
(subgoal top-goal =name (action: decide) (goal (action: move object: $objects)))	goal-1 with subgoal goal-3
(goal (action: decide reason: =reason))	*no matches*

Thee following would be illegal tests against the goal tree:

TESTS:	REASON FOR ILLEGALITY:
(top-goal (action: recognize object: ball))	top-goal not a legal type
(supergoal =goal =goal2 (action: move) (object: ball) (place: =place))	Illegal format
(goal =of =name (action: go place: =place1))	<of-goal> should not be present in type *goal*

4.2.5. Left-Hand Side Function Calls

There are a variety of external function calls available to the GRAPES user. They can be arbitrarily nested in the patterns (as above), or may be used on the *outer level* as in:

```
(*success =goal)
```

which would return *t* or *nil* depending on whether the goal which bound to =goal was successful at the time this condition was tested, or whether the goal was not successful. If *t* is returned, the other conditions of the production continue through the matching process. If *nil* is returned, the production fails.

Function calls can be built-in GRAPES *functions like *pattern and *current-goal, or they can be user defined *functions loaded in as a special module[9]. In addition, regular LISP functions can be used on the left-hand side. If an atom begins with "*" and does not have a function definition, GRAPES calls the function without the "*". This can be used quite effectively when matching. For instance: (*not (*equal =result1 =result2)) will match only when =result1 and =result2 bind to different values.

There are two types of outer-level function calls: those that return a boolean value, and those that return a pattern. If a function call returns a pattern, it is matched against working memory. If it returns a boolean variable, the production will fail when the boolean value is *nil*. Function calls are described in more detail in section five.

4.2.6. Negative Tests

If a left-hand side pattern is preceded by a "-" symbol, then that pattern is taken to be a negative test. If the pattern is a working memory tests, this means that the given production will match only if no elements in working memory match to the pattern. If the pattern is a goal test, then the given production will match only

---

[9] see a description of the LISP module in appendix seven

if there is no goal in the goal-tree (or no supergoal or subgoal, depending on the goal type) which matches the specification. If the pattern is an outer-level function call which doesn't return a pattern, then the production will match only if the call returns *nil*.

As an example:

```
(p p-2
   action: get-tense
   word: =word
   tests:
     - (is-a =word plural)
==>
   (is-a =word singular))
```

which says that if there is nothing in memory which says that the word is plural, then it is singular.

4.2.7. Partial Matching

If a left-hand side pattern is preceded by a "?" symbol, then that pattern is matched as a positive test. If the matching succeeds, the variables are treated like they are in a positive test. If the pattern does not match, the pattern is considered to be a negative test, and those variables only mentioned in the pattern are not given bindings. This facility can be used to produce partial matching and ordering of productions based on their degree of match. In addition, the partial match test can be used in conjunction with right-hand side patterns using the same variable. For instance, if *use-name* were a production:

```
(p use-name
   action: get-name
   tests:
       (name-list $names)
     ? (has-name =object =name)
       (object-list $obj1 =object $obj2)
==>
   (*remove 1)
   (name-list $names =name) )
```

it might add the name of an object to a name-list. If the object already had a name, it would use that name and if the object did not have a name, a new name would be made. See section below for information on how a new name is made.

4.3. The Right-Hand Side

The action side is an ordered list of zero or more GRAPES expressions. Once conflict resolution has picked one instantiation of a particular production and all the variables on the left-hand side have bindings, that production's right-hand side is evaluated from top to bottom. The right hand side can put elements into working memory, put goals into the goal tree, or execute LISP functions. These are each described in the next

three sections.

4.3.1. Additions to Working Memory

If a GRAPES pattern is found on the right-side side of a production, and its first element is not a *goal-word* (supergoal, subgoal, goal) or a function call, then it is evaluated and stored in working memory. All variables in the pattern which had bindings on the left-hand side, are evaluated according to those bindings. If a variable was not bound on the left-hand side, then a binding is made for the variable, and it is bound using the *bind function[10]. The new value looks like the variable but is guaranteed to be unique. For instance, if the variable =var was found on the right-hand side without a binding, then a value which looks something like @var1 would be bound to that variable.

4.3.2. Additions to Goal Memory

All right-hand side patterns beginning with a goal-word are added to the appropriate place in the goal tree. The goal patterns look very similar to the patterns which match to them. Each goal is of the form:

```
( <type> [ <of-goal> [ <name> ]]
         <parameters> [ <goal1> [ <goal2> ... ]] )
```

<type> Either *goal*, *subgoal*, or *supergoal*.

<of-goal> A goal name which will have as a new supergoal or subgoal.

<name> The default value is the current goal or the goal specified above, if it is a nested goal specification.

<name> The name of the goal with the given parameters.

<parameters> The list of goal parameters. For example:

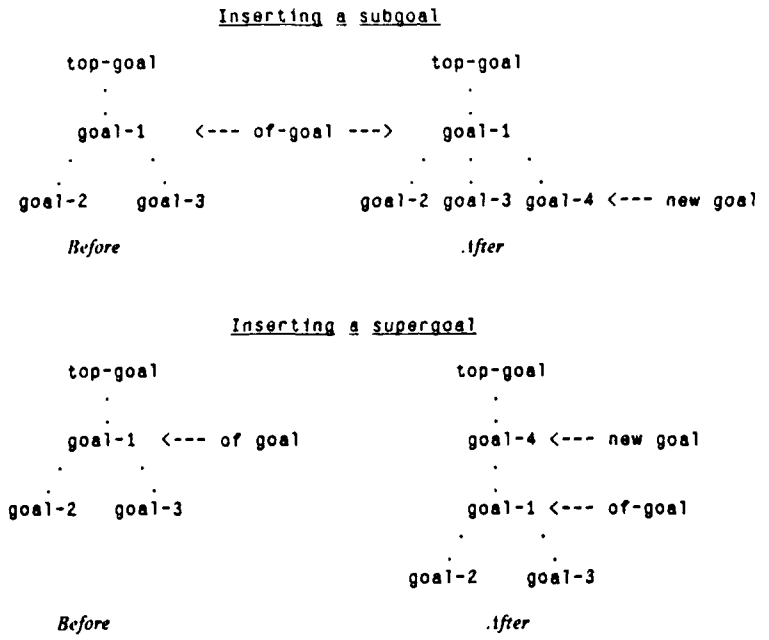
```
(action: write
 function: *name)
```

<goal n> if given, each is a nested goal specification.

However, the goal words are interpreted differently. If the goal <type> is *goal* or *subgoal*, the new goal is set as an immediate subgoal to <of-goal> or or an immediate subgoal to the current goal if <of-goal> is not specified. If goal specifications are being nested, <of-goal> defaults to the goal above the goal specification being defined. If the goal type is *supergoal*, the goal is set as a new supergoal to <of-goal>. Examples of

[10] see section 6

inserting goals are given below.



With any goal, if the goal <name> already exists, an *or* node is pushed onto the supergoal of <name>. That is, the old goal and its subgoals are replaced by the new one. The *or* node will be a new goal which has as its subgoals all of the old subgoals (thus preserving the old state of the goal tree through this new node).This enables the system to remember its previous moves and act accordingly. Putting goals into various places in the goal tree can sometimes alter the status of already successful goals. For instance,when inserting a subgoal (see above), the of-goal would become an active goal even if it was previously successful. The new goal adds additional contraints to the satisfaction of the of-goal. All subgoals of a given goal must be successful for the goal to be successful. For more information on the goal structure see section 3.

4.3.3. Right-Hand Side Function Calls

If a GRAPES pattern has a *function as its first element, then that function is evaluated according to the bindings of the current instantiation. If that function yields a working memory element, then that element is stored in working memory. The function may also return a boolean value. If the value is *non-nil*, the rest of the right-hand side is evaluated. If the value is *nil*, the rest of the right-hand side is ignored[11] . Right-hand

[11] this does not mean the production won't match, since matching has already taken place

side functions, like left-hand side functions, can be defined in LISP or user defined[12].

[12] see section 4.1.5

5. The Recognize-Act Cycle

When running, the system goes through a series of *recognize-act* cycles. In each cycle the system goes through the following steps:

1. GRAPES looks through the goal tree and finds a goal which it wants to solve. This goal becomes the focus of attention during this cycle, and is called the *current-goal*.

2. GRAPES looks through its production set and, based on the contents of working memory, the current goal, and the remaining goals in the goal tree, chooses a subset of the productions which it thinks is relevant to solving the current-goal. This subset is called the *conflict set*.

3. A process known as *conflict resolution* is applied, whose purpose is to single out one production from the conflict set which will be used to try and solve the current-goal. If there is no such resulting production, then GRAPES decides that it cannot solve the current-goal, and we go back to step 1. Otherwise, the resulting production is called the *current-production*.

4. GRAPES *fires* the current-production. If we take the view that each production is a miniature program, then this means that GRAPES runs the current program. When this happens, many changes in the program environment can occur. Data can be added to or deleted from working memory, and new goals may be inserted into the goal tree.

5. GRAPES goes back to step 1, beginning a new cycle in the context of a new program environment.

Each of these steps is treated in detail in the next few sections.

5.1. Finding the Current Goal

The current goal is always the deepest and left-most node in the goal tree which has not yet reached *success* status. The current goal reaches *success* status when :

1. All of the current goal's subgoals have reached *success* status, or

2. A production explicitly states that the goal is successful.

The current goal is matched against the patterns in the production's parameter list using the matching algorithm. The current goal must have an action which is an atom identical to that of the production action attribute. In addition, all of the attributes specified in the production must be present in the goal, also.

5.2. Matching - Determining the Conflict Set

The GRAPES matching algorithm is basically a *data-flow* type. When being compiled, each left hand side pattern makes a function definition. These function definitions are used to match against working memory and the goal tree. The match network is *pre-compiled* in the sense that most patterns are not actually matched during the matching part of the cycle. The matching is done before-hand, when working memory and goal memory are being defined. The matches for each pattern are updated each time an item is put into working memory or goal memory. This item *flows* through the match network creating variable bindings as it goes.

At the match part of the production cycle, the matches of each pattern are analyzed to see if they are consistent with goal memory and working memory. All productions patterns are examined at this time, thus the productions are being tested in parallel. Some GRAPES patterns require *dynamic matching*[13] . That is, their possible matches can't be determined ahead of time. All LISP function calls and some goal tests have to be matched dynamically.

When all matching is done, a given production may have matched in several ways each of these possible production possibilities is an *instantiation*. The process of deciding among these instantiations is called *conflict resolution.*

5.3. Conflict Resolution

GRAPES uses five basic rules to decide among competing instantiations. Goal memory makes the first rule possible. The second rule is a heuristic used to make the system more efficient. The third rule comes from the working memory structure. The flexibility of GRAPES patterns makes the fourth rule possible. The final rule is used only as a last resort.

These rules for ordering instantiations are done in sequence. Those that follow the action test are passed on to the refraction test. Those that pass that test are sent on to the next test, and so on. In the end, one production instantiation remains (thus GRAPES has no parallelism in its firing mechanism). The rules are listed in order below.

5.3.1. Action Specification

Each production must have an action. Only instantiations of productions with the same action as the current goal can apply. This restriction can be used by a GRAPES programmer to isolate productions into *subroutines*, groups of productions which apply to a particular task. If all actions are equal, GRAPES will look

---

[13] see section 4

at all the productions. The possibility of subroutines can be taken to an extreme, however. A different action for each production would make the conflict set Every time. This would an uninteresting production system for most applications.

5.3.2. Refraction

Each production, once used at a particular goal, cannot fire with the same same bindings at the same goal. This is to prevent the system from getting into one production loops (though longer loop cycles are possible).

5.3.3. Recency

Each working memory element and goal element has a *time tag* associated with it, which denotes when it was put into working memory. Those instantiations which matched to the most recent working memory elements are preferred. Although GRAPES contains, at this time, no decay of working memory elements, this instantiation ordering strategy approximates decay to some extent. Elements most recently entered into memory probably have more relevance to the problem at hand.

5.3.4. Specificity

Each GRAPES pattern has a specificity associated with it. The production goal parameters and the tests against working and goal memory are all considered for resolution on a basis of specificity. A few simple rules are followed when deciding the specificity:

1. A list is more specific than a constant.

2. A constant is more specific than a variable.

3. A regular (=) variable is more specific than a segment variable.

Specificity allows the more specific productions to apply when in competition with more general productions. Since productions are never removed from production memory, this can be a very important ordering strategy[14].

[14] *this becomes even more important when a system includes learning mechanisms, like the next version of GRAPES*

5.3.5. Randomness

If, after applying all the ordering rules to the production instantiations, a list of possible instantiations still remains, the system picks one at random.

5.4. Firing a Production Instantiation

When one instantiation of a particular production is chosen, the *recognize* part of the cycle is over, and the system performs actions. Various actions can be performed:

1. Additions to working memory

2. Additions to goal memory (subgoals or inserted goals)

3. Removal from working memory

4. Calls to pre-defined functions (input and output fall under this category)

5. Calls to LISP (user defined functions or LISP primitive functions)

Each of these acts is described in more detail in section 4 on production right hand side actions. It must be stressed that GRAPES productions' right hand sides are **not** executed in parallel. The sequences of actions is a linear description of what the system should do.

5.5. Continued Processing

The recognize-act cycle is performed again and again until the system is explicitly halted, or until the top-goal's status is determined. At each cycle a new environment is formed which the system must adjust to. The essential parallel nature of production systems is partially avoided in GRAPES, where goal structure plays a key role. Proper use of the GRAPES control structure can make programming in a production system language much easier.

6. User Functions

GRAPES has a set of predefined functions which aid the user in writing productions. The names of each of these functions begin with a "*" character. All atoms which are the first element of a list and have a "*" as the first character in their name are considered to be calls to external LISP functions. A summary of user functions is in appendix II.

6.1. General Functions

These functions are general purpose functions and can be used either on the left-hand side or on the right-hand side of a production. Also, they can be nested within working memory tests, etc.; they do not need to be outer-level function calls.

*bind
  form: (*bind =var value)
  args: =var: A variable name.
        value: A lisp expression, possibly containing GRAPES
          variable names.
  synopsis:
    Creates variable bindings. If the given variable is
    unbound, then its binding is set to the given value. If the
    variable has already been bound to the given value, then
    nothing happens, but the binding operation is still considered
    to be successful. If the variable has already been bound, but
    to a different value, then the binding is unsuccessful.
  returns:
    The specified value, if the binding was successful. If the
    binding is unsuccessful, and we are on the right-hand side of
    a production, then "nil" is returned. An unsuccessful binding
    on the left-hand side of a production causes that production
    instantiation to fail.
  side effects:
    The given variable becomes bound if the binding
    was successful.

*current-goal
  form: (*current-goal)
  args: none.
  synopsis:
    Gets the name of the current goal.
  returns:
    The name of the current goal.

*length
  form: (*length [arg1 arg2 ...])
  args: Each arg(i) is optional and may be any expression.
  synopsis:
    Creates a list of unbound variable names. The number of
    variables in the list is the same as the number of arguments
    passed to the function. This is useful in conjunction with the
    *pattern and *match functions.
  returns:
    A list of unbound variable names.
    As an example, if the variable "$argset" is bound to the list
    "(arg1 arg2 arg3)", then the call:
      (*length $argset)
    would return something like
      (=V0001 =V0002 =V0003).

*match
  form: (*match pattern expression)
  args: pattern: An expression containing constants, variables, etc.
    expression: Any expression.
  synopsis:
    Tries to match the given pattern to the given expression.
  returns:
    "t" if the match was successful. Otherwise, returns "nil".
  side effects:
    If the match was successful, then all variables in the
    given pattern which were previously unbound are now given
    bindings.

*new
  form: (*new =symbol)
  args: =symbol: A variable name.
  synopsis:
    Creates new symbols. The =symbol given should be an unbound
    variable name.
  returns:
    A new atom which looks like the given symbol name. For
    example, "(*new =arg)" might return "@arg1".
  side effects:
    The given variable is bound to the new symbol, using
    the *bind function.

*pattern

form: (*pattern [arg1 arg2 ...])
args: Each arg(n) is optional and may be any expression.
synopsis:
Creates a pattern from the list of its arguments. This is usually used in conjunction with the *match function.
returns:
A pattern, which is usually used to do a *match. For example, if "$argset" is bound to the list "(arg1 arg2)", then the call:
(*pattern (*length $argset) =var (*length $argset))
would return something like
(=V0001 =V0002 =var =V0003 =V0004).

*subgoals
form: (*subgoals [=goal-name])
args: =goal-name (optional): The name of a goal in the goal tree.
synopsis:
Gets the list of subgoals of the specified =goal-name. If no goal name is given, gets the list of subgoals of the current goal.
returns:
A list of subgoals, or "nil".

*supergoal
form: (*supergoal [=goal-name])
args: =goal-name (optional): The name of a goal in the goal tree.
synopsis:.
Gets the name of the supergoal of the specified =goal-name. If no goal name is given, gets the supergoal of the current goal.
returns:
The specified supergoal, or "nil".

6.2. Left-Hand Side Functions

These functions are usually used on the left-hand side of a production, since they test for properties or conditions existing during the execution of a GRAPES program.

*gmethod
form: (*gmethod [goal [var]])
args: goal (optional): The name of a goal in the goal tree.
var (optional; goal must be specified if var is specified): A variable name.
synopsis:

Gets the method associated with the given goal. If no goal is specified, it is assumed to be the current goal. If "var" is given, then the method of the specified goal is bound to "var" (using the *bind function).

returns:

The method associated with the specified goal, or "nil" if there is no associated method.

*methodp

form: (*methodp [=goal] =method)

args: =goal (optional): The name of a goal in the goal tree.
=method: Any expression which is a legal method for a goal.

synopsis:

Matches against the method of a goal. If no =goal is specified, then it is assumed to be the current goal.

returns:

"t" if the method associated with the given goal matches the given expression; otherwise, returns "nil".

*success

form: (*success =goal)

args: =goal: The name of a goal in the goal tree.

synopsis:

Tests for success status of a goal.

returns:

"t" if the the named goal has been successful; otherwise (if the goal has failed or is still currently active) returns "nil".

6.3. Right-Hand Side Functions

These are functions which are used on the right-hand side of a production, since they perform actions which alter properties of the GRAPES program environment.

*input

form: (*input [arg1 arg2 ...])

args: Each arg is an arbitrary expression. See below.

synopsis:

Reads data from the terminal. Each arg is treated as they are for *output, except that unbound GRAPES variables are given values which must be input by the user.

returns:

"t".

*mapgoal
  form: (*mapgoal =var list [=result] goal-spec)
  args: =var: A variable name.
        list: A list of arbitrary expressions.
        =result (optional): A variable name.
        goal-spec: A goal specification, in the same form as a goal
          specification would ordinarily appear on the RHS of
          a production.
  synopsis:
      Inserts a list of goals into the goal tree. "=var" is bound
      successively to each member of the given list. One goal
      is created (according to the given goal specification) on each
      iteration; thus, it is usually desired to have "=var" appear
      somewhere within the goal specification. If "=result" is
      specified, then the list of the names of the goals created
      is bound to it using the *bind function.
  returns:
      "t".

*mapstore
  form: (*mapstore list)
  args: list: A list of expressions.
  synopsis:
      Each expression is stored in working memory. Each such expression
      should be a list; any which are not lists are ignored.
  returns:
      "t".

*method
  form: (*method [goal] expression)
  args: goal (optional): The name of a goal in the goal tree.
        expression: An arbitrary expression.
  synopsis:
      Sets the method of the given goal to be the given expression.
      If no goal is given, it is assumed to be the current goal.
  returns:
      "t".

*output
  form: (*output [arg1 arg2 ...])
  args: Each arg is an arbitrary expression which is treated
        as specified below.
  synopsis:

Each arg(i) is treated according to the following specifications:

1. If the arg is a string, it is printed.

2. If the arg is a list, then every element in the list is "*output"ed.

3. If the arg is an integer, then the next arg is "*output"ed that number of times.

4. If the arg is "/", then a carriage return is printed.

5. If the arg is "#", then the next arg is an expression which evaluates to an integer, and a tab to that cursor position is performed.

6. If the arg is "@", then the next arg is evaluated and its result is printed.

7. If the arg is a GRAPES variable, then it is evaluated and its result is printed.

8. All other args are just printed.

returns:
"t".

*pop
form: (*pop [status])
args: status (optional): Either "success" or "failure".
synopsis:
Controls the flow of the program through the goal tree by explicitly setting the status of the current goal. Goals can be declared to be a success or a failure. If no status is specified, it is assumed to be "success".
returns:
"t".

*redo
form: (*redo goal)
args: goal: The name of a goal in the goal tree.
synopsis:
Specifies that the given goal is to be redone. This is done by removing the previous status of the goal and declaring it to be active.
returns:
"t".

*remove
  form: (*remove n)
  args: n: An integer.
  synopsis:
    Removes an element from working memory. The "nth" working
    memory element (as specified on the left hand side of the
    production) which actually matched during the current
    production firing is the one which is removed. Negative
    working memory tests and "?" tests which failed do not count
    (since they did not match).
  returns:
    "t".

*replace
  form: (*replace =new oldList oldElement newElement)
  args: =new: An unbound variable to hold the new list
    oldList: A list.
    oldElement: An expression in oldList.
    newElement: A new expression.
  synopsis:
    This is the GRAPES substitute function. It binds =new to list
    which is the same as oldList except that every occurance of
    oldElement is replaced by newElement. This replacement is done
    over all levels of the list.

6.4. About Function Calls in GRAPES

Any atom which is in the function position of a list, and which begins with the character "*", is considered to be a call to an external LISP function. This function call may be one of those which have been pre-defined by the interpreter (as outlined in the previous three sections), or they may be user-defined LISP functions which have been *slurp*ed in by the user. Both types are treated the same by GRAPES.

6.4.1. Calls to Common LISP Functions

When GRAPES sees a function name of the form **name*, it looks for a definition of the function called **name*. If there is no such function definition, then it looks for the function called *name*. Thus, the user can use most LISP functions (those which are defined in Franz LISP) inside productions by attaching a "*" to the beginning of the function name. For example, the following is a legal function call in GRAPES:

```
(*not (*equal =list1 (*cdr =list2))).
```

6.4.2. Defining Your Own Functions

The interpreter assumes that all external functions are written in such a way that all arguments are evaluated; that is, they must be defined as *lambdas* or *lexprs*, or *macros* which expand to *lambdas* or *lexprs*. This means that they can be fairly general function definitions. They can be called from within productions or from within other function definitions. The user can also use any of the pre-defined GRAPES functions from within his own function definitions.

Writing your own user functions to be used by GRAPES is easy; they are simply written as if they were going to be used by any LISP program. Don't worry about passing GRAPES variables to LISP functions; the GRAPES interpreter does some pre-processing before functions are called so that this will work. For example, the function call

```
(*not (*equal =list1 (*cdr =list2)))
```

will be pre-processed so that, at the time the call is actually made, it will look something like:

```
(not (equal '(a b c d)) (cdr '(b c d)))).
```

One convention used in pre-processing is that all GRAPES variables which are bound at the time the function is called are replaced by their bindings during pre-processing. Unbound GRAPES variables, however, are not replaced by anything[15] . This allows a variable name to be passed to a user function. Then, a (*bind =var value) may be used from within the user function in order to create a new variable binding.

A few final words of caution are in order here. First, be careful with functions that do not evaluate their arguments (like *nlambdas*) and with functions that evaluate their arguments in special ways. For example, it is a bad idea to use the following type of function call from within a GRAPES production:

```
(*or (*eq =list1 =list2)
     (*setq =list1 =some-value)).
```

In general, *setq*s may give problems, since assigning a variable a LISP value does not affect the value which GRAPES knows about. To get around this, use the **bind* function. Finally, be careful not to use functions in the wrong places. For example, using a **mapstore* on the left-hand side of a production will not necessarily cause an error, but will have some very strange side effects.

[15] no (*new =symbol) is generated

7. Using the Interpreter

The GRAPES interpreter is similar to the LISP interpreter that it is embedded within. The GRAPES interpreter allows only commands which are defined at the top level[16]. Calls to functions not defined in GRAPES result in an error. If the user would like to execute LISP commands in addition to GRAPES commands, he must type (QUIET!!) or QUIET!! to the GRAPES monitor[17]. This will make almost all Franz Lisp commands available (though the interaction is still through the GRAPES interpreter). QUIET mode is used to load in LISP files for auxillary systems not defined as modes[18].

GRAPES commands which take no arguments can simply be typed as an atom. For instance:

```
(ptrace)
```

can be typed as:

```
ptrace
```

with the same effects. This feature is also available in QUIET mode. However, only GRAPES functions can be typed this way.

First a number of commands useful for defining a production environment will be described, then a function for actually starting the system is given. Mechanisms for stopping the system and exiting GRAPES are looked at next, followed by a description of commands to reset initial conditions. Finally a group of useful pretty-printing functions are described, which make the GRAPES productions, goals, and working memory accessible to the user.

7.1. Defining the System

These functions help to set up a GRAPES programming environment.

slurp
  form: (slurp [file name] [file name] ...)
  arguments: Each file contains GRAPES productions or system commands
  synopsis:
    The slurp command will define the production set found in each file.
    The default file extension is ".grp", so that (slurp junk) will
    define productions found in the file "junk.grp". If no file name

---

[16] these are listed in appendix 1

[17] This may be the default, depending on what version you are using

[18] See mode command, section 7.1

is given or the file given cannot be opened, slurp will prompt for another file name. The file may contain calls to other system commands as well. It is often useful to contain calls to "setgoal" and "setwm" within a ".grp" file, so that they will be defined automatically when the file is slurped in.

returns:
"Production Environment Defined."

{}
form: { <comment> }
synopsis: The GRAPES comment characters. All characters between the curly brackets will not be read by the interpreter.
returns: nothing

setgoal
form: (setgoal '(action: <action> <other parameters>))
argument: Must be a legal goal.[19]
For example, a legal call is:
(setgoal '(action: write
arg1: arg-list
result: result-list))
synopsis:
The setgoal command sets the value of the "top-goal". The top-goal is the default goal used to start the system.
returns:
A message concerning the success or failure of the call.

setwm
form: (setwm '(<list1> <list2> <list3> ...))
arguments: Each argument is a list representing one working memory element.[20]
synopsis:
Setwm sets the contents of working memory, erasing any elements currently there. Setwm expects a list of lists. Each list is stored as a single element in working memory. If setwm is given an atom to store, it simply makes a list containing that atom.
returns:
"Working memory defined."

---

[19] see section 3.

[20] see section 2

mode

form: (mode <special module name>)

arguments: A predefined special module

synopsis:

Loads in a special module. GRAPES version 4 offers a LISP module, which contains GRAPES's internal knowledge of LISP programming. Version 5 of the interpreter offers a learning module, described in later sections. In version 5, if the command *(setp special-modules <modules>)* is issued, <modules> will become the new list of accessible external packages. The learning module is one such module which has been added.

returns:

"O.K."

7.2. Starting and Stopping the System

start

form: (start [goal [working-memory]])

arguments: The goal defaults to the *top-goal* and working memory defaults to nil, which assumes the current working memory.

synopsis:

The start command begins running the production set in the context of the given goal and with the given working memory. The goal may be any goal which has been defined by the system (or the top-goal). If the given goal is a list, then (setgoal goal) is performed automatically. If wm is non-nil, then (setwm wm) is performed automaticlly.

returns:

An error indicating that the top-goal was never defined or (after running the production system) a message indicating the success or failure of the top-goal.

stop

form: (stop)

arguments: none

synopsis:

Stops the production system from running and returns the user to the top level of the GRAPES interpreter.[21] .

returns:

nothing

---

[21] This function does a Franz Lisp (reset)

exit
  form: exit or (exit)
  arguments: none
  synopsis:
    The exit command is used to leave the GRAPES system. This command must be typed at the top-level[22].
  returns:
    nothing (it never returns)

7.3. Restarting the System

greset
  form: (greset)
  arguments: none
  synopsis:
    The greset command clears the goal structure and sets the top goal to that which was given in the last call to setgoal.
  returns:
    A message concerning the success or failure of the call.

wmreset
  form: (wmreset)
  arguments: none
  synopsis:
    The wmreset command sets the contents of working memory to that given in the last call to setwm. If no call to setwm has been given, then working memory is cleared.
  returns:
    "Working memory defined."

preset
  form: (preset)
  arguments: none
  synopsis:
    The preset command deletes the current production set. To start another production run, the slurp command must be used.
  returns:
    "O.K."

clear

---

[22] not during a production run - use stop for that purpose

form: (clear)
arguments: none
synopsis:
The clear command removes the production set from the environment, and deletes both the goal structure and working memory. This command does not disable greset or wmreset from salvaging the previous top goal or working memory.
returns:
okay

7.4. Printing Useful Information

pp
form: (pp [production-name])
arguments: A legal GRAPES production name currently defined to the system or no arguments.
synopsis:
Pretty prints a production to the terminal. The form used in pretty printing is the form which the system is internally using (not simply a copy of the input production specification). This function is useful to new GRAPES users who are not familiar with GRAPES's standard production format. Productions do not have to be in this form, but they are much easier to read, and they make it easier for interaction between GRAPES programmers. This function will print out all of the productions currently defined if given no arguments.
Returns:
The pretty printed production(s).

ppwm
form: (ppwm)
arguments: none
synopsis:
Pretty prints the contents of working memory in the form:
"WORKING MEMORY: "
<contents of working memory>
"END"
returns:
END

ppcurrentgoal
form: (ppcurrentgoal)
arguments: none
synopsis:

Pretty prints the current goal.

returns:

"endgoal"

pptopgoal

form: (pptopgoal)

arguments: none

synopsis:

Pretty prints the top goal.

returns:

"endgoal"

ppgoal

form: (ppgoal [goal])

arguments: none or a goal name

synopsis:

Pretty prints the given goal. If no goal is given, the system tries to print the current goal. If no current goal is defined, then it will try to print the top goal. If the top goal is not defined, then an error is returned and no goal is printed.

returns:

"endgoal"

8. Debugging

Production sets are rarely written correctly the first time through. For this reason, GRAPES has a few useful debugging mechanisms. When used in conjunction with the trace package, production systems can be interactively debugged with the mechanisms given.

Extensive documentation may help when trying to keep a system bug free. The GRAPES comment characters are the curly brackets {}. Any characters found between a set of curly brackets will be ignored by the GRAPES interpreter.

8.1. About the Monitor and QUIET Mode

The GRAPES monitor is entered when the GRAPES system is loaded into LISP. The system runs embedded in the Franz LISP environment. However, the monitor traps all commands and checks to see if they are legal GRAPES function calls (these are listed in Appendix I). Legal GRAPES commands with no arguments can be typed as atoms[23]. The QUIET function turns off function checking so that LISP commands can be used. Some GRAPES commands go by the same name as certain LISP functions. These functions cannot be accessed. For instance, *help* gives help on GRAPES not on LISP. *Load* must be used instead of *siurp* and *old-pp*!! must be used instead of *pp*[24].

8.2. Breaking

At any time the user can type control-C. This is handled as an interrupt and can be continued with the command *go*. The user can break from a break, and so on. Each subprocess is stored. The last subprocess to break from is the subprocess which is restarted when *go* is typed.

Breaking can be used extensively in debugging, when goals, working memory, and productions might need to be printed in the middle of a production run. Breaking can also be set automatically with the trace package (see section 9).

---

[23] see section 7

[24] this command is used for production pretty printing

8.3. GRAPES Debugging Commands

All GRAPES debugging commands are functions with no arguments.[25].

eset — This function returns the current conflict set (in list form). The conflict set is a list of the names of the instantiations which will undergo conflict resolution. If a particular production had more than one possible instantiation, several copies of the production name might be in the list. The eset command is good for analyzing what productions competed for a match to the current working memory and goal memory.[26]

resolve — This function traces the conflict resolution process responsible for picking the current production. If recency was used in deciding among competing productions, the recency of all the working memory items matching each production will be pretty printed. If specificity was also used, the summed specificity of each production will be displayed. If neither strategy singled out one production, a message is printed saying that the conflict was resolved through a random choice[27].

bindings — When a particular instantiation is decided upon, its bindings can be displayed using the *bindings* command. This prints the list of variables in the production followed by the value which the variable matched. During the matching process, the variable may have bound to several possible configurations of values. Only the final bindings of the variables are shown.

---

[25] thus they can be typed without the parenthesis

[26] see section 5

[27] section 5 has more details on the conflict resolution strategies

9. Special Packages

The GRAPES interpreter includes a trace package, a break package, and a photo package. Since the break package and the trace package are used together, they will be described together. Then, the photo package will be introduced.

9.1. Tracing and Breaking

Both goals and productions can be traced. The productions are traced when they fire, and the goals are traced when they become the current goal. A sample trace is shown on the following page.

This trace shows a hypothetical production run in which both goal tracing and production tracing are used. The "| |" symbols indicate the depth in the goal tree. "5" is the firing number[28] . "Use-hand-coding-knowledge" is the name of the current production, and "goal-1" is the name of the current goal. All tests against working memory and goal memory are shown, with the values they bound to. LISP calls on the left-hand side are shown before evaluation. The right hand side actions are also shown. Methods attached to goals are given. Additions and deletions from working memory are also shown. The evaluations of user functions are shown if they return a GRAPES pattern, otherwise they do not show up on the trace. Goals inserted into random places in the tree are listed separately from "ordinary" subgoals. The immediate supergoal and the immediate subgoals of each inserted goal are also listed. If a goal is *redone*, the tracer shows the goal tree structure leading to that goal. The goal parameters of any goal are also traced (in the example these would be "action" and "argument". All variables are replaced by the values to which they bound.

9.1.1. Starting the Trace

The default trace includes only the firing number, the production name, the goal name, and the depth markers. All other options can be set with the following two commands:

ptrace
  form: (ptrace [/switch1 [/switch2 ....]] [pname1 [pname2 ...]])
  arguments: A set of switches and a set of productions
  synopsis:
    Performs a production trace on the given productions. If no productions are given, then all productions are traced. Switches are optional and apply to all production names which follow them. Switches can be made to apply to only selected productions by enclosing the switches and the production in parenthesis.

---

[28] simply the number of productions which have fired so far

```
| | | goal-1
| | | .   action: solve-by-hand
| | | .     arg: @refined-result1
| | | endGoal
| | | | 5) use-hand-coding-knowledge
| | | | .  TESTS:
| | | | .     (has-relation result1 (all path-from) (start))
| | | | .  METHOD:
| | | | .     hand-search.
| | | | .  INSERTED into wm:
| | | | .     (has-relation term1 member (list1))
| | | | .     (policy (avoid-repeats refined-result1))
| | | | .  REMOVED from wm:
| | | | .     (has-relation result1 (all path-from) (start))
| | | | .  SUBGOALS created:
| | | | .     (goal-5
| | | | .         (action: make-by-hand
| | | | .          arg: list2
| | | | .          value: (start)) )
| | | | .     (goal-6
| | | | .         (action: repeat-until-failure
| | | | .          condition: @goal2
| | | | .          do: @goal3) )
| | | | .  INSERTED into goal tree:
| | | | .     (@goal2
| | | | .         (action: find
| | | | .          arg: term1
| | | | .          constraint: (not tried)) )
| | | | .        Subgoal of: goal-6.
| | | | .     (@goal3
| | | | .         (action: perform-operations
| | | | .          args: (@goal4 @goal5)) )
| | | | .        Subgoal of: goal-6.
| | | | .        Subgoals are: (@goal4 @goal5).
| | | | endProduction
```

Figure 9-1: A Sample Production Trace

If a production trace is in effect, all information about the production(s) is printed.

returns:

A list of all productions being traced.

gtrace

form: (gtrace [/switch1 [/switch2 ...]] [action1 [action2 ...]])

arguments: A set of switches and set of goal actions.

synopsis:

Performs a goal trace on all goals with the given actions. If no actions are given, then all goals are traced. Switches are treated the same way as in production tracing. If a goal trace is in affect, all information about the goal(s) is printed.

returns:

A list of all goal actions being traced.

9.1.2. Stopping the Trace

If tracing is no longer desired, it can be turned off with the following commands:

puntrace

form: (puntrace)

arguments: none

synopsis:

Returns production tracing to the default mode (only production names and firing numbers printed).

returns:

A compact list of all productions untraced.

guntrace

form: (guntrace)

arguments: none

synopsis:

Returns goal tracing to the default mode (only goal names and depth markers printed).

returns:

A compact list of all goal actions untraced.

untrace

form: (untrace name1 [name2 [name3 ...]])

arguments: Each name is either a production name or a goal action.

synopsis:

Untraces select goals and productions.

returns:

"O.K."

9.1.3. Trace Switches

Each trace command can set a number of switches. These switches tell the trace routine when to perform certain traces and when to perform a break.

/break1

Does a break just before a production fires. A message like:

Break before production <pname>.
Type 'go' to continue.

[BREAK <n>]

will be printed. Where <pname> is the production being traced, and <n> is the current break level (see section 8.2). Each time the given production or productions fire, a break is executed.

/break2

Does a break just after a production fires. It is used the same way as the /break1 switch. A message will be printed like

Break after production <pname>
type 'go' to continue.

[BREAK <n>]

/break

Sets break-points at the given goals if a "gtrace" is being done. Sets break-points both before and after the given productions if a "ptrace" is being done.

/after

The /after switch is used to state that tracing is not to begin until a given production has fired. For example,

(ptrace /after p5)

will trace all productions, but only after p5 has fired. If the argument to /after is a list, then tracing will begin after any one of the given productions has fired.

/until

The /until switch is similar to the /after switch, except it specifies that tracing is to halt when one of the given productions fires. It is

useful in conjunction with the /after switch. For example,

```
(gtrace /after p5 /until (p7 p8))
```

will trace all goals as soon as p5 fires, and will discontinue tracing when either p7 or p8 fires.

9.2. Photo

The photo package records a GRAPES session. The photo file will contain all interaction with GRAPES from the time the *photo* command is issued to the time the *unphoto* command is issued. The photo file is *automatically terminated when GRAPES is exited. Nested photo sessions are not allowed.* The photo commands are given here:

photo

form: (photo [filename[.ext]])

argument: A legal file name

synopsis:

Writes all output to the given file. The default file extension is ".log", and the default file name is "photo". For example,

```
(photo test)
```

would send all output to a file named "test.log".

returns:

"PHOTO recording initiated: <filename>"

unphoto

form: (unphoto)

arguments: none

synopsis:

Stops recording the GRAPES session.

returns:

"PHOTO recording terminated."

The trace package and the photo package interface nicely, so that traces of production runs can be recorded in full detail.

10. Learning Mechanisms

This section of the document describes how to use the GRAPES learning mechanisms to acquire new productions. This facility is only offered in GRAPES version 5.

There are two types of learning mechanisms currently available to GRAPES users. These are *composition* and *proceduralization*. *Composition* involves collapsing productions which fired at goals in a given section of the goal tree. *Proceduralization* reduces the number of long-term memory retrievals made on a production left-hand side. Together composition and proceduralization form *compilation*. This section does not intend to cover all the aspects of the GRAPES learning mechanisms. However, many useful learning production sets can be written with the material presented here[29].

10.1. Proceduralization

The GRAPES proceduralization mechanism follows John Anderson's ACT model quite closely. In GRAPES, as in ACT, knowledge is found in two separate and fundamentally different forms. The first type of knowledge is *declarative knowledge*,knowledge about facts, which is stored propositionally. The second type of knowledge is *procedural knowledge*, knowledge about processes, which is stored as production rules. Declarative knowledge can be in long-term memory or in working memory. In the ACT model, working memory is the active part of long-term memory. In GRAPES, the memory elements do not have associated activation levels, so the user must specify exactly what propositions are in long-term memory and which are in working memory. The items in long-term memory are a subset of the items in working memory. Productions can only add to working memory and goal memory. Long-term memory items are added using the "setwm" command. Long-term memory is not destroyed between production runs, so knowledge can be accumulated, making learning modelling easier.

10.1.1. Interpretive Productions

When procedures have not been encoded as production rules, they reside in memory in the form of declarative procedures. Productions which access declarative procedures are said to be *interpretive*. These interpretive productions work over a large range of circumstances and tend to be very general. However, long-term memory retrieval is very slow, so using interpretive productions can be costly in any information processing system. Proceduralization provides a way of reducing a production's access to long-term memory by building the long-term memory information into the production directly. In the current implementation, proceduralized productions are matched before other productions, producing an obvious speedup in

[29] see appendix three for a more detailed description of the learning parameters

execution time. In addition, proceduralized productions faster to match since they contain less left-hand side tests than their corresponding interpretive versions. Note that proceduralized productions will fire in place of the original productions under the appropriate circumstances. However, the original interpretive productions are still available to the system in situations where no proceduralized rules are applicable.

10.1.2. Finding What to Proceduralize

The GRAPES proceduralization mechanism is invoked automatically when the parameter *learning-now* is set to "t"[30]. All productions which access long-term memory fall prey to the proceduralization mechanism. A production is *not* proceduralized if it involves setting goals whose action specifies planning. Those goals which involve planning must be defined by the user (though some default values exist). In GRAPES it is assumed that productions which set goals to plan are fundamentally interpretive in nature and should not be proceduralized.

10.1.3. Forming Special Case Rules

If a production has been chosen to be proceduralized, its last successful instantiation is retrieved and searched for tests against long-term memory. Each variable which bound to a proposition in long-term memory or part of a proposition in long-term memory is replaced by the object that it bound to. This incorporates the information contained in the instantiation directly into the new production. A host of special case rules can be formed this way. Note that variables are replaced on both the right-hand and left-hand sides.

10.1.4. Production Strength

GRAPES version 5 possesses *strength classes*. Each production has an associated strength value (whose default is 1.0). Productions in the highest strength class are matched first. If none of these productions match, the next strength class is considered. This procedure is repeated until all successful matches within a strength class have been found, or until there are no more productions left to match. When a production is proceduralized, its strength becomes greater.

10.2. Composition

The GRAPES composition mechanism collapses pieces of the goal tree to form a plan for doing some large action. The user has control over which pieces of the goal tree he wishes to include in the composition process.

Before loading a production set, the user must set some learning parameters which tell the system which

---

[30] This is the default value

productions to compose and when to compose them. These parameters can be set using the *setp* command (see section on parameters).

10.2.1. When to Compose

The *learning-actions* parameter holds a list of actions which tell GRAPES when to start composing. When a goal with a *learning action* is popped as a success, GRAPES examines the goal tree beneath the goal for possible compositions. If a *learning action* goal is declared successful, it signifies the completion of a problem or a subproblem.

10.2.2. Where to Start

The *flag-actions* parameter is set to a list of actions which tell GRAPES where to start composing. Goals with *flag actions* specify a task which needs to be accomplished. These goals are called *composition headers*. The group of productions which fired at goals beneath the composition header form a macro-operator for performing the specified task. The composition mechanism will combine this group of productions to produce a single rule which has the effect of the group and therefore accomplishes the task.

10.2.3. Where to End

The *non-learned-actions* parameter holds a list of actions which tell GRAPES where to stop composing. These goals often involve checking and re- examining, so they are automatically set as subgoals on the right-hand side of the composed production.

GRAPES must have a way to link the macro-operators together to accomplish a sequence of tasks. The composition mechanism sets each composition header goal as a subgoal on the right-hand side of the preceding composed production. In this way, one macro-operator can set a goal which will be performed by another macro-operator.

10.2.4. The Composed Left-Hand Side

Productions which have been chosen for composition can have various pieces of their left-hand sides included in the final composed production. All tests against working-memory are included unless they matched against an element inserted by another production in the group being composed. Likewise, all goal tests are included unless they match to goals in the group being collapsed. This group of goals is called the *goal block*. User function tests are only included in the left-hand side of the composed production if their component parts are not part of any working-memory element added by productions in the group being composed. Parameters for the composed production are exactly those which matched the specification for the

composition header goal.

10.2.5. The Composed Right-Hand Side

The right-hand side of the composed production contains all actions relevant to performing the task specified by the composition header goal. In general, all additions to working memory are included and all additions to the goal tree calls to LISP are not included. A goal addition is included if it is a *non-learned* goal or a specification of a new task, both described above. Goals which were inserted outside of the goal block are also included.

Function calls are included in the composed production if the function is on the *physical-user-functions* list. These functions usually perform operations on an external memory[31].

If a production's action is a member of the *planning-actions* list, its right-hand side is not included in the composed production's right-hand side. Planning productions are only concerned with intermediate results, not crucial to the action of the macro-operator.

10.2.6. Changes in the Production Appearance

The composition process makes some changes in the patterns which are finally included in the macro-operator. String variables are replaced by equivalent sequences of regular variables. Variables are renamed in a consistent way across elements in the composed production's left-hand side. A variable is replaced by its binding if the binding appears as a constant in another match element. On the action side of the composed production, *mapstore and *mapgoal's are replaced by sequences of working-memory and goal elements. Finally, if compilation is taking place, proceduralization will take effect before composition, deleting some elements from consideration before composition begins.

Successful use of the GRAPES learning mechanisms is dependent on how well the component productions are written, and how intelligently the learning parameters are set. If both are done well, new and useful production rules result.

---

[31] see *write in appendix seven

10.3. An example of learning

This example shows how a simple goal tree and set of productions can be used to proceduralize and compile new productions.

The goal tree:

```
                top-goal
                action: write
                function: second
                args: list3
                output: list1
                     .
                     .
                goal-1
                action: code-relation
                function: second
                arg: list1
                .         .
               .           .
              .             .
        goal-2        goal-3
        arg: @list1   arg: list1
```

The working-memory:

```
(has-relation list1 second list3)
(calculated-by first car)
(calculated-by end cdr)
(isa list3 function-argument)
```

The long-term memory:

```
(calculated-by first car)
(calculated-by end cdr)
```

The parameter settings:

```
flag-actions            =   (code-relation)
learning-actions        =   (write)
non-learned-actions     =   ()
planning-actions        =   ()
physical-user-functions =   ()
```

The productions:

```
(p write-function
   action: write
   function: =function
   args: =arg
   output: =output
   tests: (has-relation =output =function =arg)
==>
   (goal
      (action: code-relation
       relation: =function
       arg: =output)))

(p find-second
   action: code-relation
   relation: second
   arg: =lis
   tests:
        (has-relation =lis second =result)
==>
    (has-relation =lis first =lis2)
    (has-relation =lis2 end =result)
    (goal
        (action: get-function
         arg: =lis2))
    (goal
        (action: get-function
         arg: =lis)))

(p see-car
   action: get-function
   arg: =arg
   tests: (has-relation =arg first =result)
          (calculated-by first =function)
          (calculated-by =result =expr)
==>
   (calculated-by =arg (=function =expr))
   (*pop success))

(p found-cdr
    action: get-function
    arg: =arg
    tests:  (has-relation =arg end =lis)
            (isa =lis function-argument)
            (calculated-by end =function)
==>
   (calculated-by =arg (=function =lis))
    (*pop success))
```

The GRAPES run of this environment, using the learning mechanisms:

```
[*] (mode learn) ; The learning module must be
        ; loaded before the production set
O.K.

[*] (slurp example)


   write-function
```

```
  find-second
  see-car
  found-cdr

  *** Top goal defined. ***

  *** Working and Long Term Memories defined. ***

  *** Parameter(s) Set. ***

Production Environment Defined.

[*] ptrace

  write-function
  find-second
  see-car
  found-cdr

[*] gtrace

  write
  code-relation
  get-function

[*] start

top-goal
.  action: write

.    function: second
.    args: list3
.    output: list1
endGoal
| 1) write-function. [1]
| . TESTS:
| .    (has-relation list1 second list3)
| . SUBGOALS created:
| .    (goal-1
| .       (action: code-relation
| .        relation: second
| .        arg: list1) )
| endProduction
```

```
|
|
| goal-1
| . action: code-relation
| .   relation: second
| .   arg: list1
| endGoal
| | 2) find-second. [1]
| | . TESTS:
| | .   (has-relation list1 second list3)
| | . INSERTED into wm:
| | .   (has-relation list1 first @list1)
| | .   (has-relation @list1 end list3)
| | . SUBGOALS created:
| | .   (goal-2
| | .     (action: get-function
| | .      arg: @list1) )
| | .   (goal-3
| | .     (action: get-function
| | .      arg: list1) )
| | endProduction
| |

| |
| | goal-2
| | . action: get-function
| | .   arg: @list1
| | endGoal
| | | 3) found-cdr. [1]
| | | . TESTS:
| | | .   (has-relation @list1 end list3)
| | | .   (isa list' function-argument)
| | | .   (calculated-by end cdr)
| | | . INSERTED into wm:
| | | .   (calculated-by @list1 (cdr list3))
| | | endProduction
| | Goal successful.
| |
| | goal-3
| | . action: get-function
| | .   arg: list1
| | endGoal
| | | 4) see-car. [1]
```

```
| | | . TESTS:
| | | .   (has-relation list1 first (@ list1)
| | | .   (calculated-by first car)
| | | .   (calculated-by (@ list1 (cdr list3))
| | | . INSERTED into wm:
| | | .   (calculated-by list1 (car (cdr list3)))
| | | endProduction
| | Goal successful.
| Goal successful.
Goal successful.

Defining Proceduralized Production:
   pre-found-g-2-1:


(p pre-found-g-2-1 2.0
   action: get-function
   arg: =arg
   tests:
     (has-relation =arg end =lis)
     (isa =lis function-argument)
  ==>
   (calculated-by =arg (cdr =lis))
   (*pop success) )

Defining Proceduralized Production:
   pre-see-g-3-2:


(p pre-see-g-3-2 2.0
   action: get-function
   arg: =arg
   tests:
     (has-relation =arg first =result)
     (calculated-by =result =expr)
  ==>
   (calculated-by =arg (car =expr))
   (*pop success) )

Defining Composed Production:
   comp-find-g-1-3:
```

```
(p comp-find-g-1-3 2.0
   action: code-relation
   relation: second
   arg: =lis2
   tests:
      (has-relation =lis2 second =result1)
      (isa =result1 function-argument)
  ==>
   (has-relation =lis2 first =@lis11)
   (has-relation =@lis11 end =result1)
   (calculated-by =@lis11 (cdr =result1))
   (calculated-by =lis2 (car (cdr =result1)))
   (*pop success) )
```

END-- Top Goal Successful.

Notice that the last rule is a compiled rule. The GRAPES learning mechanisms do compilation, rather than pure composition as a default learning strategy. The compiled rule above states that if you want to get the second element of a list and the list is a function argument, use the *car* of the *cdr*– of the list[32].

[32] in many LISPs this function is called cadr

Appendix I
Summary of System Commands

Command	Arguments	Description
bindings	0	Displays the variable bindings for the current production instantiation.
clear	0	Removes production set and clears working and goal memories.
cset	0	Displays the names of productions in the current conflict set.
greset	0	Resets the top goal.
gtrace	n	Traces all goals with the specified action [All goals].
guntrace	0	Turns off all goal tracing.
help	n	Displays help on the given topic. If no arguments are given, a menu is displayed.
mode	1	Loads in the module associated with the given name.
photo	1	Records a terminal session.
pp	1	Pretty prints the given production.
ppcurrentgoal	0	Pretty prints the current goal.
ppgoal	0 or 1	Pretty prints the given goal. [current goal - top goal].
pptopgoal	0	Pretty prints the current top goal.
ppwm	0	Pretty prints the present contents of working memory.

preset	0	Deletes the current production set.
ptrace	n	Traces the given productions.
puntrace	0	Turns off production tracing.
resolve	0	Displays the strategies used to decide on the current production instantiation.
setgoal	1*	Sets the top goal.
setwm	1*	Sets working memory to the argument list.
slurp	n	Reads in a set of production files. [prompts]
start	0,1,or 2*	Starts the system with the given goal [top goal] and the given working memory [present wm].
stop	0	Stops the current GRAPES run and returns to the top level.
unphoto	0	Terminates the current photo session.
untrace	1 .. n	Stops tracing the given productions or all goals with the given action.
wmreset	0	Resets the current working memory to that given in the last "setwm".

Summary of Trace Switches

Switch	Arguments	Description
/after	n	Sets tracing to begin after a

		production or production list.
/break	n	Sets breakpoints before and after a set of productions, or at all goals with a given action.
/break1	n	Sets breakpoints before a set of productions.
/break2	n	Sets breakpoints after a set of productions.
/until	n	Sets tracing to stop after a production or a production list.

NOTE: *n* specifies that any number of arguments can be given. "[ ]" indicates a default value. "*" indicates that function evaluates its arguments (functions which take no arguments are not "*" 'ed).

Appendix II
Summary of User Functions

Command	Arguments	Use	Description
*bind	(var value)	L & R	Explicitly binds <var> to <value>.
*current-goal	none	L & R	Gets the name of the current goal.
*gmethod	([goal [var]])	L	Gets the method of <goal> and binds it to <var>.
*input	n	R	Binds each arg. to data read in from the terminal.
*length	n	L & R	Creates a list <n> long of unbound variables.
*mapgoal	(var list [res goal-spec)	R	Successively binds <var> to <list>, creating goals according to the <goal-spec>. Binds result to <res>.
*mapstore	n	R	Stores each expression in working memory.
*match	(pat expr)	L & R	Tries to match <pat> to <expr>.
*method	([goal] expr)	R	Sets the method of <goal> to <expr>.
*methodp	([goal] method)	L	Matches the method of <goal> to <method>.
*new	(sym)	L & R	Creates a new symbol which looks like <sym>.

*output	n		R	Prints information to the terminal.
*pattern	n		L & R	Creates a pattern out of its arguments.
*pop	(status)		R	Sets current goal's status to <status>.
*redo	(goal)		R	Declares <goal> active.
*remove	n		R	Removes the working memory element from memory which matched the <n>th positive pattern on the LHS.
*subgoals	(goal)		L & R	Get's the subgoals of <goal>.
*success	(goal)	u	L	Tests if <goal>'s status is success.
*supergoal	(goal)		L & R	Get's <goal>'s immediate supergoal.

Note: Default values are not given here. See section 6. for a more complete description. Optional arguments are given in "[ ]", while function arguments are given in "< >" when referred to in the descriptions. *n* means that one or more arguments are possible. *L* means that the function can be used on the left-hand side of productions. *R* means that the function can be used on the right-hand side.

Appendix III
New Functions and Changes

A number of changes have been made to the implementation of GRAPES version 4. These changes do not effect the performance of the system on old GRAPES programs. The following functions are simply imply added features of the new GRAPES interpreter:

setwm

Now in addition to having a working memory, GRAPES has a long-term memory. Long-term memory consists of copies of working memory items which have been defined by the user to be of long-term status. To put something into long-term memory one simply precedes it with a "*" when doing the setwm command. For example: (setwm '((has-relation john father rob) * (isa john man))) will load the first proposition into working memory only and the second proposition into working memory and long-term memory. Long-term memory is kept between calls to setwm. Long-term memory can only be reset by doing (clear) or (exit) commands.

pp

When the command "(pp learned)" is given, all learned productions are printed to the terminal.

mode

Learning is invoked by issuing the command "(mode learn)". This will set some default parameter values. See the parameter list for these defaults.

show

(show <parameter>) shows a parameter and its current value. (show) prints all parameters and their associated values.

setp

(setp <parameter> <value>) Sets the value of a user-accessible parameter. Some amount of type checking is done.

ppltm

Prints the contents of long-term memory.

comp-p

(comp-p [goal [file]]) Shows a production composition at a given goal and prints results to a file. The goal defaults to the top-goal and the

file defaults to the terminal. The new production is not defined to the system.

prc-p

(prc-p [goal [file]]) Shows a production proceduralization at a given goal and types results to a file. The goal defaults to the top-goal and the file defaults to the terminal. The new production is not defined to the system.

def-comp

(def-comp [goal [file]]) Does a production composition at a given goal and prints the results to a file. The goal defaults to the current goal and the file defaults to the terminal. The new production is defined to the system.

def-prc

(def-prc [goal [file]]) Does a production proceduralization at a given goal and prints the results to a file. The goal defaults to the current goal and the file defaults to the terminal. The new production is defined to the system.

cmpl-p

(cmpl-p [goal [file]]) Does a proceduralization at all goals beneath a given goal and composes the results. The final production is send to a file. The new production is defined to the system, though none of the intermediate proceduralizations are defined.

*compose

(*compose [goal (file) goal ....]) Does an explicit production composition from the right hand side of any production. Composes at each goal in the argument list. When a list is encountered, it sends the compositions at each of the goals following to the file name in parentheses. All goals must exist in the goal tree and all files must be able to be opened or an error results. The goal defaults to the current goal and the file defaults to the terminal.

*proceduralize

(*proceduralize [goal (file) goal ...]) Does an explicit proceduralization from the right-hand side of a production. Proceduralizes the productions which fired at each goal in the argument list. When a file name is encountered, it sends the proceduralizations at each of the goals following the file name to the given file.File names must be in parenthesese. All goals must exist in the goal tree and all files must be able to be

opened or an error results. The goal defaults to the current goal and the file defaults to the terminal.

Appendix IV
User-Accessible Parameters

GRAPES version 5 has a set of parameters which are available to the user. Parameter settings are often made in the same file which holds the production rules.

special-modules

A LISP association list whose key is the name of the module and whose data is the access-path to that module. The default value is currently ((learn . "sys$sysdevice:[farrell.comp]glearn.o") (lisp . "sys$sysdevice:[farrell.comp]glisp.o")).

default-strength

A real number representing the strength for the productions whose strength is not given. The default is 1.0.

cycle-trace

This is a list if three possible values, which specifies what conflict resolution information should be printed every cycle. When "recency" is included in the list and recency was used in conflict resolution, time tags of matching working memory elements are displayed. If "specificity" is included in the list and specificity was used to decided what production to fire, working memory and goal specificities will be displayed. If "randomize" is in the list and a production is chosen at random, a message will be printed which tells that a random production was chosen. The default value for this parameter is "nil", which means that no conflict resolution information will be printed on every cycle.

user-accessible-parameters

This parameter is a list of the currently accessible parameters. The default value is: "(planning-actions special-modules flag-actions non-learned-actions learning-actions user-accessible-parameters physical-user-functions learning-now production-creation-trace default-strength user-function-check cycle-trace)". Notice that *user-accessible-parameters* is itself available for change. This facility is available so that the user can give himself access to more

parameters. If the user defines a new package which has some global variables which need to be set, then he can set those variables with GRAPES commands.

production-creation-trace

When this parameter has the value "t", new productions will be printed as they are defined. If it has the value "nil" they will not be printed. The learning module assigns this parameter a default value of "t".

planning-actions

A list of actions whose right-hand sides will not be included in the composition process. These are also actions which tag a production as fundamentally interpretive. Planning goals often involve adding intermediate results to working memory, and often match on long-term information not directly connected to facts in the task domain. It is precisely these goals which when collapsed into other goals and compiled make a useful plan for achieving a given action.

user-function-check

This parameter is to protect the user. When set to "t" all illegal GRAPES commands generate an error. When set to "nil", the user may use LISP commands also. The default value is dependent on the version and site.

physical-user-functions

A list of right-hand side user functions which will always be included in the composition process. Usually such functions involve making a change in an external memory.[33]

non-learned-actions

A list of actions for goals which will not be included in the composition process. These are usually goals which involve checking, which must be done all of the time. The compiled production will set a goal with the "non-learned" action. Thus, this goal should be a *terminal goal* in the piece of the goal tree accessed by the composition mechanism. That is, goals beneath

---

[33] See the section on the LISP package for an example of an external memory.

the goal with the non-learned action will not
be included in the compiled production.

flag-actions

A list of actions for goals which will be *composition headers*. Composition headers are goals which are the top goals in the composed block of the goal tree. They signal where to start composing.

learning-actions

A list of actions which tell the system when to begin composing. When a goal with a learning action is popped as a success, the composition mechanism is activated.

learning-now

This parameter becomes "t" when the learning module is loaded. When set to "nil", the system will not learn.

Appendix V
A Sample Production Set

The following is a production set for the Tower of Hanoi problem. The goal recursion strategy is already coded into the GRAPES architecture, so that goal bookkeeping productions are not needed. Notice that because of GRAPES's perfect memory for propositions, the two error checking productions are not used. If GRAPES produced working memory failures while doing this problem, the last two productions would help keep the system from making incorrect moves.

```
{ This production fires when a single disk cannot be moved. The goal
  must be broken up into subproblems. }

(p make-subproblems
   action: move
   object: =object1
   to: =pegY
   tests:
          (has-part =object1 =part1 =part2)
          (on =part2 =pegX)
          (smaller-than =part1 =part2)
          (isa =pegZ peg)
          (*not (*equal =pegZ =pegX))
          (*not (*equal =pegZ =pegY))
==>
   (goal
       (action: move
        object: =part1
        to: =pegZ))
   (goal
       (action: move
        object: =part2
        to: =pegY))
   (goal
       (action: move
        object: =part1
        to: =pegY)))
{ This production moves a disk. It assumes that all single disks can be
  moved.}

(p move-disk
   action: move
   object: =object1
   to: =pegA
   tests:
        (on =object1 =pegB)
        (isa =object1 single-disk)
==>
        (*remove 1)
        (on =object1 =pegA)
        (*pop success))
```

```
{ This production is optional. It makes sure that we are not moving
  a disk which is not the smallest on the peg. Note that conflict resolution
  will favor this production over the simple move-disk if a move is wrong.}

(p cannot-move-disk
    action: move
    object: =object1
    to: =pegA
    tests:
         (on =object1 =pegB)
         (isa =object1 single-disk)
         (on =object2 =pegB)
         (smaller-than =object2 =object1)
==>
    (*pop failure))

{ This production is optional also. It makes sure that we are not placing
  a disk on a peg which already has a disk on it which is larger. This
  production will also be preffered in conflict resolution if a move is
  illegal. }

(p illegal-move
   action: move
   object: =object1
   to: =pegY
   tests:
        (on =object1 =pegX)
        (isa =object1 single-disk)
        (on =object2 =pegY)
        (smaller-than =object2 =object1)
==>
    (*pop failure))

{ The top goal is to move pyramid-A from where it is to peg-3. }

(setgoal
     '(action: move
       object: pyramid-A
       to: peg-3))

{ The facts about pegs, disks, and pyramids are encoded in working memory.
  The initial situation is also coded in working memory, though GRAPES allows
  one to access auxillary memories on the left and right hand sides of
  productions using LISP functions. }

(setwm '((has-part pyramid-A pyramid-B disk-A)
         (has-part pyramid-B disk-C disk-B)
         (isa peg-1 peg)
         (isa peg-2 peg)
         (isa peg-3 peg)
         (isa disk-A single-disk)
         (isa disk-B single-disk)
         (isa disk-C single-disk)
         (smaller-than disk-B disk-A)
         (smaller-than disk-C disk-A)
         (smaller-than pyramid-B disk-A)
         (smaller-than pyramid-B pyramid-A)
         (smaller-than disk-C pyramid-B)
         (smaller-than disk-C disk-B)
         (on disk-A peg-1)
         (on disk-B peg-1)
         (on disk-C peg-1)))
```

Appendix VI
A Sample Run

This is an actual GRAPES photo file made with the "photo" command. The productions "slurp"ed are listed in appendix 4.

```
PHOTO recording initiated: ghanoi.log

[*] (slurp ghanoi)  ; The default extension is .grp


   make-subproblems
   move-disk
   cannot-move-disk
   illegal-move

   *** Top goal defined. ***

   *** Working memory defined. ***

Production Environment Defined.

[*] pptopgoal    ; Print the top goal
top-goal
  action: move
    object: pyramid-A
    to: peg-3
endgoal

[*] ppwm         ; Print working memory
WORKING MEMORY:
   (has-part pyramid-A pyramid-B disk-A)
   (has-part pyramid-B disk-C disk-B)
   (isa peg-1 peg)
   (isa peg-2 peg)
   (isa peg-3 peg)
   (isa disk-A single-disk)
   (isa disk-B single-disk)
   (isa disk-C single-disk)
   (smaller-than disk-B disk-A)
   (smaller-than disk-C disk-A)
   (smaller-than pyramid-B disk-A)
   (smaller-than pyramid-B pyramid-A)
```

```
  (smaller-than disk-C pyramid-B)
  (smaller-than disk-C disk-B)
  (on disk-A peg-1)
  (on disk-B peg-1)
  (on disk-C peg-1)
END

[*] ptrace   : Trace productions firing

 make-subproblems
 move-disk
 cannot-move-disk
 illegal-move

[*] gtrace   : Trace goal information

 move

[*] start    : Start the system. The top goal and current working memory
        ; are the defaults.

top-goal
.  action: move
.    object: pyramid-A
.    to: peg-3
endGoal
| 1) make-subproblems.
| . TESTS:
| .    (has-part pyramid-A pyramid-B disk-A)
| .    (on disk-A peg-1)
| .    (smaller-than pyramid-B disk-A)
| .    (isa peg-2 peg)
| .    (*not (*equal peg-2 peg-1))
| .    (*not (*equal peg-2 peg-3))
| . SUBGOALS created:
| .    (goal-1
| .       (action: move
| .        object: pyramid-B
| .        to: peg-2) )
| .    (goal-2
| .       (action: move
| .        object: disk-A
| .        to: peg-3) )
```

```
| .   (goal-3
| .      (action: move
| .       object: pyramid-B
| .       to: peg-3) )
| endProduction
|
|
| goal-1
| .  action: move
| .   object: pyramid-B
| .   to: peg-2
| endGoal
| | 2) make-subproblems.
| | . TESTS:
| | .   (has-part pyramid-B disk-C disk-B)
| | .   (on disk-B peg-1)
| | .   (smaller-than disk-C disk-B)
| | .   (isa peg-3 peg)
| | .   (*not (*equal peg-3 peg-1))
| | .   (*not (*equal peg-3 peg-2))
| | . SUBGOALS created:
| | .   (goal-4
| | .      (action: move
| | .       object: disk-C
| | .       to: peg-3) )
| | .   (goal-5
| | .      (action: move
| | .       object: disk-B
| | .       to: peg-2) )
| | .   (goal-6
| | .      (action: move
| | .       object: disk-C
| | .       to: peg-2) )
| | endProduction
| |

| |
| | goal-4
| | .  action: move
| | .   object: disk-C
| | .   to: peg-3
| | endGoal
| | | 3) move-disk.
```

```
| | | . TESTS:
| | | .   (on disk-C peg-1)
| | | .   (isa disk-C single-disk)
| | | . INSERTED into wm:
| | | .   (on disk-C peg-3)
| | | . REMOVED from wm:
| | | .   (on disk-C peg-1)
| | | endProduction
| | Goal successful.
| |
| | goal-5
| | .  action: move
| | .   object: disk-B
| | .   to: peg-2
| | endGoal
| | | 4) move-disk.
| | | . TESTS:
| | | .   (on disk-B peg-1)
| | | .   (isa disk-B single-disk)
| | | . INSERTED into wm:
| | | .   (on disk-B peg-2)
| | | . REMOVED from wm:
| | | .   (on disk-B peg-1)
| | | endProduction
| | Goal successful.
| |
| | goal-6
| | .  action: move
| | .   object: disk-C
| | .   to: peg-2
| | endGoal
| | | 5) move-disk.
| | | . TESTS:
| | | .   (on disk-C peg-3)
| | | .   (isa disk-C single-disk)
| | | . INSERTED into wm:
| | | .   (on disk-C peg-2)
| | | . REMOVED from wm:
| | | .   (on disk-C peg-3)
| | | endProduction
| | Goal successful.
| Goal successful.
|
```

```
| goal-2
| . action: move
| .   object: disk-A
| .   to: peg-3
| endGoal
| | 6) move-disk.
| | . TESTS:
| | .   (on disk-A peg-1)
| | .   (isa disk-A single-disk)
| | . INSERTED into wm:
| | .   (on disk-A peg-3)
| | . REMOVED from wm:
| | .   (on disk-A peg-1)
| | endProduction
| Goal successful.
|
| goal-3
| . action: move
| .   object: pyramid-B
| .   to: peg-3
| endGoal
| | 7) make-subproblems.
| | . TESTS:
| | .   (has-part pyramid-B disk-C disk-B)
| | .   (on disk-B peg-2)
| | .   (smaller-than disk-C disk-B)
| | .   (isa peg-1 peg)
| | .   (*not (*equal peg-1 peg-2))
| | .   (*not (*equal peg-1 peg-3))
| | . SUBGOALS created:
| | .   (goal-7
| | .     (action: move
| | .      object: disk-C
| | .      to: peg-1) )
| | .   (goal-8
| | .     (action: move
| | .      object: disk-B
| | .      to: peg-3) )
| | .   (goal-9
| | .     (action: move
| | .      object: disk-C
| | .      to: peg-3) )
| | endProduction
```

```
| |
| |
| | goal-7
| | .  action: move
| | .    object: disk-C
| | .    to: peg-1
| | endGoal
| | | 8) move-disk.
| | | . TESTS:
| | | .    (on disk-C peg-2)
| | | .    (isa disk-C single-disk)
| | | . INSERTED into wm:
| | | .    (on disk-C peg-1)
| | | . REMOVED from wm:
| | | .    (on disk-C peg-2)
| | | endProduction
| | Goal successful.
| |
| | goal-8
| | .  action: move
| | .    object: disk-B
| | .    to: peg-3
| | endGoal
| | | 9) move-disk.
| | | . TESTS:
| | | .    (on disk-B peg-2)
| | | .    (isa disk-B single-disk)
| | | . INSERTED into wm:
| | | .    (on disk-B peg-3)
| | | . REMOVED from wm:
| | | .    (on disk-B peg-2)
| | | endProduction
| | Goal successful.
| |
| | goal-9
| | .  action: move
| | .    object: disk-C
| | .    to: peg-3
| | endGoal
| | | 10) move-disk.
| | | . TESTS:
| | | .    (on disk-C peg-1)
| | | .    (isa disk-C single-disk)
```

```
| | | . INSERTED into wm:
| | | .   (on disk-C peg-3)
| | | . REMOVED from wm:
| | | .   (on disk-C peg-1)
| | | endProduction
| | Goal successful.
| Goal successful.
Goal successful.

END-- Top Goal Successful.

[*], pwm
WORKING MEMORY:
  (has-part pyramid-A pyramid-B disk-A)
  (has-part pyramid-B disk-C disk-B)
  (isa peg-1 peg)
  (isa peg-2 peg)
  (isa peg-3 peg)
  (isa disk-A single-disk)
  (isa disk-B single-disk)
  (isa disk-C single-disk)
  (smaller-than disk-B disk-A)
  (smaller-than disk-C disk-A)
  (smaller-than pyramid-B disk-A)
  (smaller-than pyramid-B pyramid-A)
  (smaller-than disk-C pyramid-B)
  (smaller-than disk-C disk-B)
  (on disk-A peg-3)
  (on disk-B peg-3)
  (on disk-C peg-3)
END

[*] unphoto : This execution was recorded using GRAPES photo package

PHOTO recording terminated.
```

Appendix VII
The LISP Module

The GRAPES LISP module is a set of top-level and user functions which are loaded automatically when the command (*mode lisp*) is issued while at the top-level. The LISP package defines a set of convenient functions for modelling expert and novice LISP programmers. This module could also be used to form the basis for an automatic LISP programming system.

When the package is loaded, all goals with an action to *write* are examined. These goals should have the name of the function being written as the value of a *function* attribute, the argument list as the value of a *args* attribute, and an atom representing the output relation under a *output* attribute. All goals in this format will produce a Franz LISP lambda definition. A <?> will represent the code for the body of the function.

VII.1. Top-level commands

ltrace
- form: (ltrace)
- arguments: none
- synopsis: Starts tracing LISP code as it is being written by the system.

luntrace
- form: (luntrace)
- arguments: none
- synopsis: Stops tracing all functions currently being written by the system.

savef
- form: (savef *file*)
- arguments: *file:* the name of a legal output file.
- synopsis: Saves in *file* all function definitions which were defined by the system.

showtable
- form: (showtable)
- arguments: none
- synopsis: Pretty-prints the contents of the LISP module's symbol table. All function names, arguments, and local variables for each function currently defined are shown. Also, each symbol is displayed along with its expansion.

VII.2. User Functions

Left-Hand Side Functions

*usep
form: (*usep *name template*)
arguments: *name:* The name of a function currently defined.
template: A list consisting of a template name followed by its arguments[34].
synopsis: Returns "t" if function *name* uses *template* for its body definition. Otherwise it returns "nil".

*writep
form: (*writep *term method*)
arguments: *term:* A LISP atom which has an entry in the symbol table.
method: An expression.
synopsis: Returns "t" if *term* is written in terms of *method* in the symbol table.

*lvarp
form: (*lvarp *vars* [*goal*])
arguments: *vars:* The name of a local variable, or a list of local variables.
goal: A goal which specifies where to begin the search for the function with the specified local variables. This goal defaults to the current goal.
synopsis: Returns "t" if *vars* are all members of the local variable list of the function found.

*gvarp
form: (*gvarp *vars* [*goal*])
arguments: *vars:* The name of a global variable, or a list of global variables.
goal: A goal which specifies where to begin the search for the function with the specified local variables. This goal defaults to the current goal.
synopsis: Returns "t" if *vars* are all global variables associated with the function found.

Right-hand Side Functions

*define

---

[34] see section on templates

form: (*define *name*)
arguments: *name:* An atom representing a function name.
synopsis: Declares *name* to be a LISP function, signalling that it is going to be written by the production set. This function is usually used to define helping functions and functions which are subproblems.

*undefine
form: (*undefine *name*)
arguments: *name:* The name of a currently defined function.
synopsis: Removes a function name from the list of functions currently being written. Equivalent to erasing the function definition from the symbol table.

*use
form: (*use *name template*)
arguments: *name:* The name of a function currently defined.
template: A list containing a template followed by its arguments.
synopsis: Uses the given template to write the body of *name*. A list of templates is given in the next section.

*examine
form: (*examine *expr*)
arguments: *expr:* A LISP expression.
synopsis: Looks at *expr* and determines its properties. *mapstore can be used to store each property as a list in working memory. The properties currently examined are: member, first, end, list, atom, dotted-pair, and length.

*eval
form: (*eval *test-name expr*)
arguments: *test-name:* An unbound variable.
expr: A LISP expression.
synopsis: Evaluates *expr* and returns a list describing the result of the evaluation. If the evaluation yields a result, the list: (result-of *test-name* is = result). If the evaluation results in an error, one of the following working-memory elements is returned:

	Error	Result
1.	Unbound variable	(result-of *test-name* is lisp-error unbound-variable)
2.	Undefined-function	(result-of *test-name* is

lisp-error undefined-function)

3. Bad arg to car (result-of *test-name* is lisp-error bad-function-arg)

4. Bad arg to signp (result-of *test-name* is lisp-error bad-function-arg)

5. Other errors (result-of *test-name* is lisp-error misc-error)

*write

form: (*write *term method*)

arguments: *term:* A LISP atom in the symbol table

method: An expression representing the expansion of *term.*

synopsis: Makes a new entry in the symbol table. Generally, *term* will will be a symbol which has no concrete code associated with it. *write will expand *term* into the more concrete terms given by *method.* In this way, the body of a LISP function is written in terms of various symbols representing major parts its body. These pieces are in turn written in terms of other pieces, and so on. These pieces are often intimately linked to the goal structure. See the sample function and symbol table for the Powerset function.

*lvar

form: (*lvar *vars* [*goal*])

arguments: *vars:* The name of a local variable, or a list of names of local variables.

goal: A goal which specifies where to begin the search for the function which will acquire the new local variables. defaults to the current goal.

synopsis: Creates any number of local variables for a function. The function is retrieved by searching the actions of successive supergoals of *goal* for an action to write. If such a goal is found, then the function name can be found as the value of the *function* attribute.

*gvar

form: (*gvar *vars* [*goal*])

arguments: *vars:* The name of a global variable, or a list of names of global variables.

goal: A goal which specifies where to begin the search for the function which will use the global variables. Defaults to the current goal.

synopsis: Creates any number of global variables to be used in a function. The function is found in a search identical to that done when

adding local variables to a function[35].

Right-hand or Left-hand Side Functions

*getgvars
  form: (*getlvars [*goal* [= *varlist*]])
  arguments: *goal:* A goal which specifies where to begin the search for
    the function which will use the global variables.
    The goal defaults to the current goal.
  = *varlist:* An unbound variable.
  synopsis: Binds = varlist to the list of global variables associated
    with the function found.

*rgvars
  form: (*rgvars [*varlist* [*goal*]])
  arguments: *varlist:* The name of a global variable, or a list of
    names of global variables.
  goal: A goal which specifies where to begin the search for
    the function which has the global variables.
    Defaults to the current goal.
  synopsis: Removes the global variables in *varlist* from the list of
    global variables associated with the function found.

*getlvars
  form: (*getlvars [*goal* [*varlist*]])
  arguments: *goal:* A goal which specifies where to begin the search for
    the function which has the local variables.
    The goal defaults to the current goal.
  = *varlist:* An unbound variable.
  synopsis: Binds = varlist to the local variable list of the function
    found.

*rlvars
  form: (*rlvars [*goal* [*varlist*]])
  arguments: *goal:* A goal which specifies where to begin the search for
    the function which has the local variables.
    The goal defaults to the current goal.
  = *varlist:* An unbound variable.
  synopsis: Removes the local variables in *varlist* from function's
    local variable list.

---

[35] see *lvar command

*function
form: (*function *name* [*goal*])
arguments: *name:* An unbound variable.
goal: A goal which specifies where to begin the search for the function name.
synopsis: Returns the name of the goal referenced by *goal*. The referencing algorithm looks at successive supergoals of *goal*, looking for a goal whose action is "write". If such a goal is found, the value of its *function:* attribute is bound to *name*.

*lisp
form: (*lisp *expr*)
arguments: *<expr:> Any expression.
synopsis: Returns "t" if *expr* is LISP code. It must be either a number, a string, or a list with a function call as its first element.

LISP templates

%iteration
form: (%iteration *list repeat*)
arguments: *list:* A list to map over, taking successive cdr's.
repeat: A piece of code representing the operation to be performed on each iteration through the loop.
synopsis: This template represents a schematic way to do an iterative procedure in LISP. The basic structure of an iterative process is often the same. This code template captures the similarities between most of the iterative procedures used in basic LISP, leaving the differences to be filled in as variables. It creates a local variable to hold the result list. Whenever a local variable is used, a LISP prog structure is automatically created.
result:

```
(prog (it-var)
  loop (cond((not list) (return it-var)))
    (setq it-var (append1 it-var repeat))
    (setq list (cdr list))
    (go loop))
```

%cdr-recursion
form: (%cdr-recursion *test term result*)
arguments: *test:* A test which will terminate the recursion.
term: An action for the terminating condition of the

recursion.

result: The recursive step.

synopsis: This template represents a schematic way to do a recursive procedure using the method of cdr or tail recursion.

result:

```
(cond((not test) 'term)
      (t result))
```

%franz-lambda-function

form: (%franz-lambda-function *name arglist body*)

arguments: *name:* A function name.

arglist: A list of arguments for the function.

body: The body of the function.

synopsis: Creates a Franz LISP lambda defintion. This template is used by the LISP module to define functions specified by goals with the action "write".

result:

```
(def name
  (lambda (arglist)
    body))
```

%code-in-loop

form: (%code-in-loop *expr*)

arguments: *expr:* A piece of LISP code.

synopsis: Creates a looping structure with *expr* as the loop body. Any local variables which are used with be placed as arguments to the prog surrounding the loop.

result:

```
(prog <local-variables>
  loop expr
    (go loop))
```

VII.3. A Sample Symbol Table

The symbol table provides a way of representing and manipulating the LISP code written by GRAPES. Each symbol is stored and then expanded into the code found in the function definitions.

These two functions:

```
(def powerset
  (lambda (list1)
    (cond ((not list1) '(nil))
          (t
           (append (powerset (cdr list1))
                   (@function1 (car list1) (powerset (cdr list1))))))))

(def @function1
  (lambda (@elt1 @result1)
    (cond ((not @result1) 'nil)
          (t
           (cons (cons @elt1 (car @result1))
                 (@function1 @elt1 (cdr @result1)))))))
```

were produced from the following symbol table:

function	args	body	local vars
powerset	(list1)	@fBody1	none.
@function1	(@elt1 @result1)	@fBody2	none.

symbol	expands to
@fBody1	(%cdr-recursion list1 @term-cond1 list2)
list2	(append @result1 @term1)
@result1	(powerset @result2)
@result2	(cdr list1)
@term1	(@function1 @elt1 @result1)
@fBody2	(%cdr-recursion @result1 @term-cond2 @term2)
@elt1	(car list1)
@term2	(cons @term3 @result3)
@term3	(cons @elt1 @elt2)
@elt2	(car @result1)
@result3	(@function1 @elt1 @result4)
@result4	(cdr @result1)
@term-cond2	nil
@term-cond1	(nil)

Appendix VIII
Formal Production Specification

The following is a formal syntactic specification for GRAPES productions, given in a modified Backus-Naur form:

production ::= "(p " p-name left-side " = => " right-side ")"
p-name ::= ATOM
left-side ::= goal-context / goal-context test-part
goal-context ::= action-parameter / action-parameter other-parameters
action-parameter ::= "action: " CONSTANT
other-parameters ::= parameter / parameter other-parameters
parameter ::= attribute value
attribute ::= labelled-atom
value ::= pattern

test-part ::= "tests: " lhs-tests
lhs-tests ::= lhs-elt / lhs-elt lhs-tests
lhs-elt ::= lhs-test-type / match-type lhs-test-type
match-type ::= "+ " / "- " / "? "
lhs-test-type ::= wm-test / goal-test / function-call

wm-test ::= list-pattern
function-call ::= "(" function-body ")"
function-body ::= function-name / function-name function-args
function-args ::= pattern-elts

goal-test ::= "(" goal-header goal-body ")"
goal-header ::= goal-type / subgoal-type / supergoal-type
goal-type ::= "goal " / "goal " g-name
subgoal-type ::= "subgoal " / "subgoal " g-args
supergoal-type ::= "supergoal " / "supergoal " g-args
g-args ::= g-name / of-goal g-name
g-name ::= CONSTANT / variable / function-call
of-goal ::= CONSTANT / variable / function-call
goal-body ::= g-parameters / g-parameters nested-goal-specs
g-parameters ::= "(" parameter-list ")"
parameter-list ::= action-parameter other-parameters / other-parameters
nested-goal-specs ::= goal-test / goal-test nested-goal-specs

right-side ::= rhs-elt / rhs-elt right-side
rhs-elt ::= wm-insertion / goal-insertion / function-call

wm-insertion :: = wm-test
goal-insertion :: = goal-test

pattern :: = pattern-elt / list-pattern
pattern-elt :: = CONSTANT / variable / segment-var / function-call
pattern -elts :: = pattern / pattern pattern-elts
list-pattern :: = "(" pattern-elts ")"
labelled-atom :: = (CONSTANT <> "action") & ":"
variable :: = "=" & ATOM
segment-var :: = "$" & ATOM
function-name :: = "*" & ATOM

ATOM is any LISP atom, except "nil".
CONSTANT is any LISP atom whose first character is not "=", "$", or "*"

Non Govt

1 DR. GERSHON WELTMAN
PERCEPTRONICS INC.
6271 VARIEL AVE.
WOODLAND HILLS, CA 91367

1 Dr. Keith T. Wescourt
Information Sciences Dept.
The Rand Corporation
1700 Main St.
Santa Monica, CA 90406

1 DR. SUSAN E. WHITELY
PSYCHOLOGY DEPARTMENT
UNIVERSITY OF KANSAS
LAWRENCE, KANSAS 66044

1 Dr. Christopher Wickens
Department of Psychology
University of Illinois
Champaign, IL 61820

1 Frank R. Yekovich
School of Education
Catholic University

Navy

1 Dr. Robert Breaux
Code N-711
NAVTRAEQUIPCEN
Orlando, FL 32813

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 DR. PAT FEDERICO
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Dr. John Ford
Navy Personnel R&D Center
San Diego, CA 92152

1 LT Steven D. Harris, MSC, USN
Code 6021
Naval Air Development Center
Warminster, Pennsylvania 18974

1 Dr. Jim Hollan
Code 304
Navy Personnel R & D Center
San Diego, CA 92152

1 CDR Charles W. Hutchins
Naval Air Systems Command Hq
AIR-340F
Navy Department
Washington, DC 20361

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. William L. Maloy
Principal Civilian Advisor for
Education and Training
Naval Training Command, Code 00A
Pensacola, FL 32508

Navy

1 CAPT Richard L. Martin, USN
Prospective Commanding Officer
USS Carl Vinson (CVN-70)
Newport News Shipbuilding and Drydock Co
Newport News, VA 23607

1 Dr William Montague
Navy Personnel R&D Center
San Diego, CA 92152

1 Ted M. I. Yellen
Technical Information Office, Code 201
NAVY PERSONNEL R&D CENTER
SAN DIEGO, CA 92152

1 Library, Code P201L
Navy Personnel R&D Center
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

1 Psychologist
ONR Branch Office
Bldg 114, Section D
666 Summer Street
Boston, MA 02210

1 Office of Naval Research
Code 437
800 N. Quincy SStreet
Arlington, VA 22217

5 Personnel & Training Research Programs
(Code 458)
Office of Naval Research
Arlington, VA 22217

1 Psychologist
ONR Branch Office
1030 East Green Street
Pasadena, CA 91101

Navy

1 Special Asst. for Education and Training (OP-01E)
Rm. 2705 Arlington Annex
Washington, DC 20370

1 Office of the Chief of Naval Operations
Research Development & Studies Branch (OP-115)
Washington, DC 20350

1 LT Frank C. Petho, MSC, USN (Ph.D)
Selection and Training Research Division
Human Performance Sciences Dept.
Naval Aerospace Medical Research Laborat
Pensacola, FL 32508

1 Dr. Gary Poock
Operations Research Department
Code 55PK
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Worth Scanland, Director
Research, Development, Test & Evaluation
N-5
Naval Education and Training Command
NAS, Pensacola, FL 32508

1 Dr. Alfred F. Smode
Training Analysis & Evaluation Group (TAEG)
Dept. of the Navy
Orlando, FL 32813

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Dr. Robert Wisher
Code 309
Navy Personnel R&D Center
San Diego, CA 92152

Navy

1 Mr John H. Wolfe
Code P310
U. S. Navy Personnel Research and Development Center
San Diego, CA 92152

Army

1 Technical Director
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Mr. James Baker
Systems Manning Technical Area
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Beatrice J. Farr
U. S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 DR. FRANK J. HARRIS
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Michael Kaplan
U.S. ARMY RESEARCH INSTITUTE
5001 EISENHOWER AVENUE
ALEXANDRIA, VA 22333

1 Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Attn: PERI-OK
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Joseph Ward
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 Dr. Earl A. Alluisi
HQ, AFHRL (AFSC)
Brooks AFB, TX 78235

1 Dr. Alfred R. Fregly
AFOSR/NL, Bldg. 410
Bolling AFB
Washington, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

2 3700 TCHTW/TTGH Stop 32
Sheppard AFB, TX 76311

Marines

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

CoastGuard

1 Chief, Psychological Reserch Branch
U. S. Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

Other DoD

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and Personnel Technology
Office of the Under Secretary of Defense for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Civil Govt

1 Dr. Paul G. Chapin
Linguistics Program
National Science Foundation
Washington, DC 20550

1 Dr. Susan Chipman
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. John Mays
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 William J. McLaurin
66610 Howie Court
Camp Springs, MD 20031

1 Dr. Arthur Melmed
National Intitute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. Andrew R. Molnar
Science Education Dev. and Research
National Science Foundation
Washington, DC 20550

1 Dr. Joseph Psotka
National Institute of Education
1200 19th St. NW
Washington,DC 20208

1 Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Non Govt

1 Anderson, Thomas H., Ph.D.
Center for the Study of Reading
174 Children's Research Center
51 Gerty Drive
Champiagn, IL 61820

1 Dr. John Annett
Department of Psychology
University of Warwick
Coventry CV4 7AL
ENGLAND

1 1 psychological research unit
Dept. of Defense (Army Office)
Campbell Park Offices
Canberra ACT 2600, Australia

1 Dr. Alan Baddeley
Medical Research Council
Applied Psychology Unit
15 Chaucer Road
Cambridge CB2 2EF
ENGLAND

1 Dr. Patricia Baggett
Department of Psychology
University of Colorado
Boulder, CO 80309

1 Dr. Jonathan Baron
Dept. of Psychology
University of Pennsylvania
3813-15 Walnut St. T-3
Philadlphia, PA 19104

1 Mr Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Liaison Scientists
Office of Naval Research,
Branch Office , London
Box 39 FPO New York 09510

1 Dr. Lyle Bourne
Department of Psychology
University of Colorado
Boulder, CO 80309

Non Govt

1 DR. JOHN F. BROCK
Honeywell Systems & Research Center
(MN 17-2318)
2600 Ridgeway Parkway
Minneapolis, MN 55413

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 DR. C. VICTOR BUNDERSON
WICAT INC.
UNIVERSITY PLAZA, SUITE 10
1160 SO. STATE ST.
OREM, UT 84057

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. John B. Carroll
Psychometric Lab
Univ. of No. Carolina
Davie Hall 013A
Chapel Hill, NC 27514

1 Dr. William Chase
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94305

Non Govt

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, Ma 02138

1 Dr. Lynn A. Cooper
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Meredith P. Crawford
American Psychological Association
1200 17th Street, N.W.
Washington, DC 20036

1 Dr. Kenneth B. Cross
Anacapa Sciences, Inc.
P.O. Drawer Q
Santa Barbara, CA 93102

1 LCOL J. C. Eggenberger
DIRECTORATE OF PERSONNEL APPLIED RESEARC
NATIONAL DEFENCE HQ
101 COLONEL BY DRIVE
OTTAWA, CANADA K1A 0K2

1 Dr. Ed Feigenbaum
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Mr. Wallace Feurzeig
Bolt Beranek & Newman, Inc.
50 Moulton St.
Cambridge, MA 02138

1 Dr. Victor Fields
Dept. of Psychology
Montgomery College
Rockville, MD 20850

1 Univ. Prof. Dr. Gerhard Fischer
Liebiggasse 5/3
A 1010 Vienna
AUSTRIA

Non Govt

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1 Dr. Alinda Friedman
Department of Psychology
University of Alberta
Edmonton, Alberta
CANADA T6G 2E9

1 Dr. R. Edward Geiselman
Department of Psychology
University of California
Los Angeles, CA 90024

1 DR. ROBERT GLASER
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Marvin D. Glock
217 Stone Hall
Cornell University
Ithaca, NY 14853

1 Dr. Daniel Gopher
Industrial & Management Engineering
Technion-Israel Institute of Technology
Haifa
ISRAEL

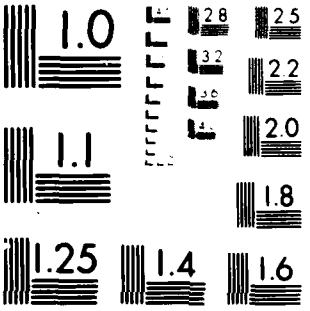
1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Harold Hawkins
Department of Psychology
University of Oregon
Eugene OR 97403

1 Dr. Barbara Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

AD-A122 360 GRAPES USER'S MANUAL(U) CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF PSYCHOLOGY R SAUERS ET AL. NOV 82 ONR-82-3 N00014-81-C-0335 2/2

UNCLASSIFIED F/G 9/2 NL



Non Govt

1 Dr. Frederick Hayes-Roth
The Rand Corporation
1700 Main Street
Santa Monica, CA 90406

1 Dr. Dustin H. Heuston
Wicat, Inc.
Box 986
Orem, UT 84057

1 Dr. James R. Hoffman
Department of Psychology
University of Delaware
Newark, DE 19711

1 Dr. Kristina Hooper
Clark Kerr Hall
University of California
Santa Cruz, CA 95060

1 Glenda Greenwald, Ed.
"Human Intelligence Newsletter"
P. O. Box 1163
Birmingham, MI 48012

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Greg Kearsley
HumRRO
300 N. Washington Street
Alexandria, VA 22314

1 Dr. Steven W. Keele
Dept. of Psychology
University of Oregon
Eugene, OR 97403

1 Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

Non Govt

1 Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85721

1 Dr. Stephen Kosslyn
Harvard University
Department of Psychology
33 Kirkland Street
Cambridge, MA 02138

1 Dr. Marcy Lansman
Department of Psychology, NI 25
University of Washington
Seattle, WA 98195

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Lesgold
Learning R&D Center
University of Pittsburgh
Pittsburgh, PA 15260

1 Dr. Michael Levine
Department of Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

1 Dr. Mark Miller
TI Computer Science Lab
C/O 2824 Winterplace Circle
Plano, TX 75075

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Donald A Norman
Dept. of Psychology C-009
Univ. of California, San Diego
La Jolla, CA 92093

Non Govt

1 Committee on Human Factors
JH 811
2101 Constitution Ave. NW
Washington, DC 20418

1 Dr. Seymour A. Papert
Massachusetts Institute of Technology
Artificial Intelligence Lab
545 Technology Square
Cambridge, MA 02139

1 Dr. James A. Paulson
Portland State University
P.O. Box 751
Portland, OR 97207

1 Dr. James W. Pellegrino
University of California,
Santa Barbara
Dept. of Psychology
Santa Barabara, CA 93106

1 MR. LUIGI PETRULLO
2431 N. EDGEWOOD STREET
ARLINGTON, VA 22207

1 Dr. Richard A. Pollak
Director, Special Projects
Minnesota Educational Computing Consorti
2520 Broadway Drive
St. Paul,MN 55113

1 Dr. Martha Polson
Department of Psychology
Campus Box 346
University of Colorado
Boulder, CO 80309

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

1 Dr. Steven E. Poltrock
Department of Psychology
University of Denver
Denver,CO 80208

Non Govt

1 Dr. Mike Posner
Department of Psychology
University of Oregon
Eugene OR 97403

1 MINRAT M. L. RAUCH
P II 4
BUNDESMINISTERIUM DER VERTEIDIGUNG
POSTFACH 1328
D-53 BONN 1, GERMANY

1 Dr. Fred Reif
SESAME
c/o Physics Department
University of California
Berkely, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Mary Riley
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 DR. WALTER SCHNEIDER
DEPT. OF PSYCHOLOGY
UNIVERSITY OF ILLINOIS
CHAMPAIGN, IL 61820

Non Govt

1 Dr. Alan Schoenfeld
Department of Mathematics
Hamilton College
Clinton, NY 13323

1 DR. ROBERT J. SEIDEL
INSTRUCTIONAL TECHNOLOGY GROUP
HUMRRO
300 N. WASHINGTON ST.
ALEXANDRIA, VA 22314

1 Committee on Cognitive Research
% Dr. Lonnie R. Sherrod
Social Science Research Council
605 Third Avenue
New York, NY 10016

1 Dr. David Shucard
Brain Sciences Labs
National Jewish Hospital Research Center
National Asthma Center
Denver, CO 80206

1 Robert S. Siegler
Associate Professor
Carnegie-Mellon University
Department of Psychology
Schenley Park
Pittsburgh, PA 15213

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Robert Smith
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

1 Dr. Richard Snow
School of Education
Stanford University
Stanford, CA 94305

1 Dr. Kathryn T. Spoehr
Pscyhology Department
Brown University
Providence, RI 02912

Non Govt

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 DR. ALBERT STEVENS
BOLT BERANEK & NEWMAN, INC.
50 MOULTON STREET
CAMBRIDGE, MA 02138

1 David E. Stone, Ph.D.
Hazeltine Corporation
7680 Old Springhouse Road
McLean, VA 22102

1 DR. PATRICK SUPPES
INSTITUTE FOR MATHEMATICAL STUDIES IN
THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CA 94305

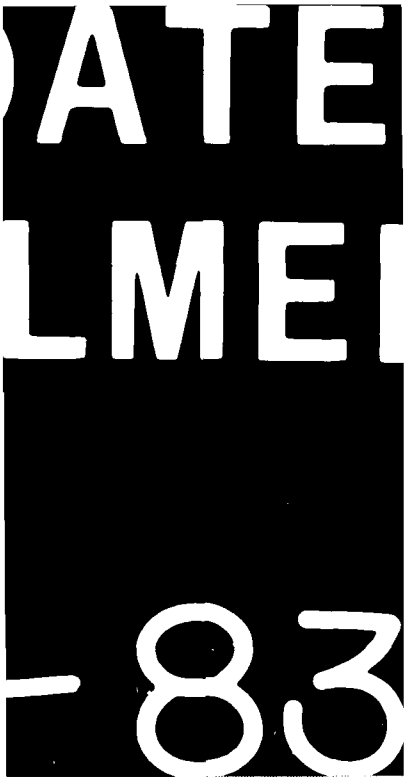
1 Dr. Kikumi Tatsuoka
Computer Based Education Research
Laboratory
252 Engineering Research Laboratory
University of Illinois
Urbana, IL 61801

1 Dr. John Thomas
IBM Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

1 DR. PERRY THORNDYKE
THE RAND CORPORATION
1700 MAIN STREET
SANTA MONICA, CA 90406

1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

1 Dr. Benton J. Underwood
Dept. of Psychology
Northwestern University
Evanston, IL 60201


ATE
LME
83