A Constructivist Approach to Artificial Intelligence Reexamined

by

Robert Matthew Ramstad

B.S. Massachusetts Institute of Technology (1991)

Submitted to the Dept. of Electrical Eng. and Computer Science in partial fulfillment of the requirements for the degrees of

Master of Science

and

Bachelor of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1992

Author ....
Dept. of Electrical Eng. and Computer Science
July 8, 1992

Certified by ....
Ronald L. Rivest
Professor, Dept. of Electrical Eng. and Computer Science
Thesis Supervisor

Certified by ....
Bruce A. Foster
Principal Software Engineer, Digital Equipment Corporation
Thesis Supervisor

Accepted by ....
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

A Constructivist Approach to Artificial Intelligence Reexamined

by

Robert Matthew Ramstad

Submitted to the Dept. of Electrical Eng. and Computer Science
on July 8, 1992, in partial fulfillment of the
requirements for the degrees of
Master of Science
and
Bachelor of Science

Abstract

"Made-Up Minds: A Constructivist Approach to Artificial Intelligence", a Ph.D. thesis by Gary Drescher (MIT, Computer Science, September 1989) and a book published by MIT Press (1991) describe a novel learning system which controls a simulated robot and gathers information about causes and effects for various actions within the *microworld* (a software simulated world) the robot inhabits. Beliefs about causality in the microworld are constructed through a learning process which is driven by the continuous updating of statistics. Each belief, or *schema*, held by the system has an associated reliability factor, and the system is able to iteratively improve both the reliability and scope of its knowledge base by revising and strengthening previously held beliefs on the basis of new statistically significant information. Schema structures are easily understood by humans — at any point in time, the amount of knowledge acquired by the system can be determined by direct examination.

Unfortunately, Drescher's system is computation- and hardware-intensive. This work documents the reimplementation of this learning system from the ideas in the thesis and book alone, using Common LISP and a general purpose UNIX hardware platform to encourage further experimentation with these ideas. Execution of the reimplementation code indicates that Drescher's results are implementation independent and directly attributable to the ideas in his published works. Results not discussed in Drescher's works were also discovered.

Thesis Supervisor: Ronald L. Rivest
Title: Professor, Dept. of Electrical Eng. and Computer Science

Thesis Supervisor: Bruce A. Foster
Title: Principal Software Engineer, Digital Equipment Corporation

Acknowledgments

I'd like to acknowledge the help of my thesis supervisors, Professor Ronald Rivest, MIT, and Bruce Foster, Digital Equipment Corporation, for their assistance, cooperation and patience throughout all phases of this thesis. The assistance of the VAX LISP group at Digital Equipment Corporation was invaluable in the early going, particularly the help of Paul Anagnostopoulos (macros), Jeff Piazza (dark secrets of LISP compilers), Walter van Roggen (code optimization) and Richard Wells (general Common LISP). Professor Pattie Maes and the MIT Media Lab Music and Cognition group have been a great bunch of people to work with over the last year. I'd like to thank Kevin O'Toole, the VI-A program staff and Marilyn Pierce for assistance throughout with various bits of thesis paperwork and administrative red tape. Robyn Kozierok, MIT Media Lab and AI Lab, assisted me greatly with an unbiased and frank opinion of a draft of this thesis. My fraternity, Beta Theta Pi, has been and continues to be a very important influence on my life, and the brothers of the MIT chapter have been instrumental in helping me keep things in focus over the years. Allison Nelson helped me through some of the worst times when I had been working on the thesis for a while and I was frustrated with the slow progress of the work. Last, but not least, I'd like to thank my mom Sheryl and sister Kristina for all their support — "Why don't you just get a job, Bob?". All of these people were instrumental in keeping me sane on the long road to the completion of this work.

I would also like to thank my favorite musical artists: David Bowie, Yes, King Crimson, The Who, Peter Murphy / Bauhaus, acoustic Neil Young, Talking Heads, Shriekback and especially The Grateful Dead. Their work has enriched my life and probably improved my code as well. (I usually listen to music when I program, it seems to keep me better focused on the task at hand.)

Contents

1 Introduction **13**

1.1 Motivation for work 15

1.2 Sources 16

1.3 The microworld 17

1.4 The schema mechanism 22

1.4.1 Schemas 23

1.4.2 Constructing new schemas via marginal attribution 24

1.4.3 Goal-directed actions 29

1.4.4 Synthetic items 30

1.4.5 Control 31

1.4.6 Summary of schema mechanism 33

1.5 Performance 33

2 Implementation **35**

2.1 Differences between this work and Drescher's 35

2.1.1 Goal-directed actions 35

2.1.2 Value 36

2.1.3 Control 37

2.1.4 Overriding conditions 38

2.1.5 Microworld 38

2.1.6 Schema mechanism 39

2.1.7 Reconciling different sources 40

2.1.8 Piagetian influence 40

2.2 Design decisions 41

2.3 Description of reimplementation code 42

2.3.1 Microworld 42

2.3.2 Schema mechanism 43

3 Results **59**

3.1 Test runs 59

3.1.1 Output 60

3.1.2 Analysis program 61

3.2 Drescher's results and schema categories 61

3.2.1 Initial schemas 62

3.2.2 Grasping schemas 62

3.2.3 Coarse visual field shifting schemas 62

3.2.4 Visual field shift limit schemas 64

3.2.5 Foveal region shift schemas 64

3.2.6 Detail shift schemas 65

3.2.7 Visual network schemas 66

3.2.8 Hand movement network schemas 67

3.2.9 Negative consequence schemas 68

3.2.10 Hand to body schemas 69

3.2.11 Seeing hand movements via coarse visual items 69

3.2.12 Seeing hand movements via detailed visual items 69

3.2.13 Further results of the CM2 implementation 71

3.3 Results confirming Drescher's work 71

3.4 Further results of the reimplementation 72

3.4.1 Shifting the gaze to see the body 78

3.4.2 Using the body as a visual position reference 78

3.4.3 Hand network constraint schemas 79

3.4.4 Visual network constraint schemas 79

3.4.5 Coarse visual shift constraint schemas 79

3.4.6 Detailed visual shift constraint schemas 80
3.4.7 Coarse to detailed visual shift constraint schemas 80
3.4.8 Detailed to coarse visual shift constraint schemas 80
3.4.9 Detailed to coarse visual shift schemas 80
3.4.10 Seeing objects in different visual regions 81
3.4.11 Hand to body constraint schemas 81
3.4.12 Hand movement against object 82
3.4.13 Backward hand movement against body 82
3.4.14 Hand movement from coarse visual to foveal region schemas . 82
3.4.15 Support data from test runs 83

4 Analysis, Discussion and Conclusions 89

4.1 Test run evidence confirms Drescher's results 89
4.2 Analysis of all results 94
4.3 Performance 97
4.4 Future directions 98

A Source code 101

A.1 Microworld simulator 102
A.2 Schema mechanism 131
A.3 Support for fixnum math 231
A.4 Support for float math 233
A.5 Result analysis 235

B Output 269

B.1 Partial output from sample run 270
B.2 Sample output from analysis program 281

Bibliography 289

List of Figures

1-1 The hand can move within a 3x3 area in the area in front of the body. 18

1-2 The visual field is 5x5 and can assume nine different orientations, for a total visual region of 7x7. 18

1-3 Five foveal regions in the center of the visual field (front, back, right, left and center) provide detailed visual information. 19

1-4 The two objects in the microworld circulate in a clockwise direction among a series of four contiguous *home positions*. 21

1-5 A sample microworld situation. The hand and the left object are in view, while the right object is out of view. 22

1-6 A basic schema. 24

1-7 A bare schema for the *grasp* action. 25

1-8 The extended result of the bare schema */grasp/* detects that */grasp/hgr* should be spun off. 25

1-9 The extended context of the schema */grasp/hgr* discovers that *tactl* improves its reliability and therefore *tactl/grasp/hgr* is spun off. . . . 26

1-10 Two schemas which predict two items separately can not chain to a schema which has both items in its context: a schema with a conjunctive result is required.. 28

2-1 The initial state of the microworld. 44

3-1 Shifting the gaze often causes an object which is within the visual field to appear at a new coarse visual item location. For example, the left hand object in this figure is moved from *vf04* to *vf14* when the glance is shifted to the left. 63

3-2 The detailed visual appearance of an object often shifts from one foveal region to another when a gaze action is taken. The hand in this figure is shifted from the rear foveal region to the center foveal region when the gaze is shifted to the rear. 65

3-3 Shifting the hand results in the hand moving to a new position. . . . 68

3-4 Shifting the hand when it is visible often causes a transition to a new coarse visual item (and possibly to a new foveal region). 70

3-5 Through a large series of schemas, the reimplementation code learns that there are three different objects which each can appear in specific regions of the microworld. 81

List of Tables

1.1 The primitive actions 19

1.2 The primitive items 20

1.3 Rules for determining synthetic item state. 32

2.1 Simple datatypes defined and used by the schema mechanism code. . 45

2.2 Rules for using counters to detect statistically relevant occurrences. . 48

2.3 Compound datatypes defined and used by the schema mechanism code. 49

2.4 Rules for extended context statistics. 51

2.5 Rules for extended result statistics. 53

2.6 Notes on some of the functions in schema.lisp 57

3.1 Further results of Drescher's CM2 implementation, part one. 72

3.2 Further results of Drescher's CM2 implementation, part two. 73

3.3 Statistics for Drescher's classifications, test runs 1–7. 74

3.4 Statistics for Drescher's classifications, test runs 8–14. 75

3.5 Statistics for Drescher's classifications, test runs 15–20. 76

3.6 Statistic summary for Drescher's classifications. 77

3.7 Statistics for new classifications, test runs 1–7. 84

3.8 Statistics for new classifications, test runs 8–14. 85

3.9 Statistics for new classifications, test runs 15–20. 86

3.10 Statistic summary for new classifications. 87

Chapter 1

Introduction

"Made-Up Minds: A Constructivist Approach to Artificial Intelligence", a Ph.D. thesis by Gary Drescher (MIT, Computer Science, September 1989) [Dre89] and a book published by MIT Press [Dre91], describe a learning system which gathers information about causes and effects while controlling a simulated robot which exists within a *microworld* (a software simulated world). The system he proposes, the *schema mechanism*, is novel in a number of ways. It falls firmly in the category of systems which learn from experience — it learns without any outside assistance. Claims have been made that these systems are "crucial to achieve successful behavior in complex, dynamic, unpredictable environments" [Mae92, p. 5] and as such they are particularly interesting systems to study. This system constructs beliefs about causality in the microworld through a learning process which is driven by the continuous updating of statistics. Each of these beliefs has an associated reliability factor, and the system is able to iteratively improve both the reliability and scope of its knowledge base by revising and strengthening previously held beliefs on the basis of new statistically significant information. The use of reliability factors and the iterative nature of improvement in the knowledge base are different from the methods found in many other learning systems, where the focus is usually on discovering *facts* about causality which are 100% reliable by exhaustive analysis of all possibilities within the problem domain itself. Also, the schema mechanism manipulates, stores and modifies schemas which represent beliefs and are easily understood by humans without using mathematical analysis tools. While many connectionist systems also have the ability to iteratively improve the reliability and scope of their knowledge bases, the derivation of the rules represented by the final configuration of the system is difficult. On the other hand, the final configuration of a schema mechanism run can be examined by a person and the highly reliable schemas can be analyzed directly to determine the amount of knowledge acquired by the system.

The schema mechanism provides control to a software simulated robot with body and hand which lives inside a microworld. Given a variety of possible actions and a vector of sensor data, the schema mechanism both attempts to reach states of high *value* (a quantitative measure of the desirability of a given state, based both upon the state itself and the range of states easily reachable from that state) and engages in behavior designed to improve its knowledge of the microworld. The schema mechanism is able to engage in a form of planning by constructing and maintaining *schemas* — structures which can be used to predict the result of taking a specified action in the current situation. The collection of schemas and certain other specialized structures comprises the knowledge the system has at each time step.

The results in the original thesis are startling. The original CM2 implementation of the schema mechanism, with virtually no initial knowledge about the microworld, manages to construct many reliable schemas. These *rules* include: how to grasp an object, how to move the hand from one position to an adjacent one, how to move the glance orientation from one position to an adjacent one, how to move an object closer to the center of the visual field so as to see its details, and then, through the construction of *goal-directed actions*, how to move the hand to any position, how to move the glance orientation to any position, how to move any object so as to see its details, and so on. In fact, the system appears to gain some idea of the concept of *objects* (through the construction of *synthetic items*), as well as some *intermodal* coordination (i.e. schemas which relate one sense to another, for example, schemas which indicate that moving the hand results in seeing the hand at the new location). The CM2 implementation was extremely successful in acquiring a large body of knowledge about the microworld. It is also notable that the progression of learning exhibited by the program closely matches the progression postulated by Piagetian child development theory, where the concept of "schemata" was first analyzed.

1.1 Motivation for work

The CM2 implementation is relatively time-inefficient, and is also computation and hardware-intensive, utilizing over four thousand processors on a CM2 Connection Machine. A better implementation, in addition to dealing with these problems, might actually be able to learn more — the results of both [Dre91] and [Dre89] are clearly constrained by available memory.

Another purpose served by a new implementation is that of verification of the ideas in the original thesis. It is unclear if the results of the CM2 implementation are solely due to the design as detailed in [Dre91], or perhaps partially due to various specific unknown aspects of the implementation. In other words, the ideas in [Dre91] and [Dre89] may not be sufficient to account for all of the results generated by the implementation. Obviously, in any system of great complexity, seemingly minor implementation decisions may have unforeseen effects on the execution of the program. A new implementation can give concrete evidence that any system built as specified in the original thesis is capable of achieving comparable results.

This thesis documents the reimplementation of this learning system from the ideas in [Dre91] and [Dre89]. Where these two sources conflict, [Dre91] takes precedence. Where the description of the schema mechanism was not sufficiently detailed, the author of the original thesis was consulted. This document is organized as follows:

- **Chapter 1**: Motivation for work, overview of the schema mechanism.
- **Chapter 2**: Modifications and additions to the ideas in [Dre91] and [Dre89] and documentation of the new implementation.
- **Chapter 3**: Results from multiple executions of the new implementation.

- **Chapter 4**: Further analysis of the results described in chapter 3, comparison of the results found by each implementation. Suggestions for future experimentation with the schema mechanism.

Perhaps the most important goal of this project, however, is to encourage other researchers to experiment with the concepts embodied in the original thesis. To this end, the new implementation is written in Common LISP and executes on a general-purpose UNIX hardware platform and was designed with efficiency in mind. It is hoped that making the system available to researchers in a form which can easily be understood, executed and modified will assist the utilization of these ideas in future research.

1.2 Sources

Gary Drescher has published many different works on the schema mechanism, notably a recent book from MIT Press titled "Made-Up Minds" [Dre91] and his Ph.D. thesis [Dre89]. Every attempt has been made to be consistent with [Dre91], as it is the most recent major work. However, the reimplementation effort started before [Dre91] was published, and therefore [Dre89] was also heavily used. Both sources are useful for interpreting the other and therefore both are very valuable. Gary Drescher was also gracious enough to answer many questions via private electronic mail and telephone, which also assisted in constructing what hopefully is an accurate view of the schema mechanism. I freely borrowed (and condensed) from each of these sources as necessary while writing this overview — it is based primarily on the material in [Dre89] but agrees with [Dre91] in all major respects. If a deeper explanation is desired, I suggest reviewing the original sources, particularly [Dre91, section 1.1 (general), section 6.1 (microworld) and chapters 3 and 4 (schema mechanism)] and [Dre89, section 1.2 (general), section 3.1 (microworld) and section 3.2 (schema mechanism)]. The overview presented here definitely emphasizes those parts of Drescher's schema mechanism which were implemented in detail, and glosses over parts of his system which were not implemented. The following chapter contains a detailed discussion of the differences between this account, the accounts in [Dre91] and [Dre89] and what was actually implemented.

1.3 The microworld

The *microworld* is a separate system which is intended to be a primitive model of the real world. The area of the microworld is modeled as a 2 dimensional 7x7 grid — all vision is from a *birds-eye* viewpoint. Objects can be placed anywhere within the 7x7 area, but only one object can exist at any single coordinate position — objects are uniform in size, and cannot rotate.

The microworld is inhabited by a simulated robot with body, single hand and visual system which can perform actions in the microworld. The initially supplied *primitive actions* allow shifting of the visual region, movement of the hand, and grasping and ungrasping of objects. Each of these actions may or may not change the state of the microworld — in particular, actions which would take the hand or glance orientation beyond the allowable range have no effect. The hand can move within a 3x3 area near the body (see figure 1-1). Similarly, there are nine glance orientations which allow viewing of any particular 5x5 area within the 7x7 microworld (see figure 1-2). The primitive actions are designed to correspond roughly with the actions available to a stationary infant, and are further described in table 1.1 [Dre89, adapted from table 3.1, p. 66].

The simulated robot receives feedback on the current state of the microworld through roughly one hundred and forty *primitive items* — items which can be on or off (binary) and are directly related to conditions in the microworld. These primitive items include indications of hand position, indications of glance orientation, coarse visual information (an object is present within the visual region of the robot), detailed visual information (when an object is near the center of the visual region, see figure 1-3), tactile indications of the presence of an object (when an object is adjacent to the hand or body), detailed tactile information (when an object is to the left of the hand), detailed taste information (when an object is in front of the body), and indications as

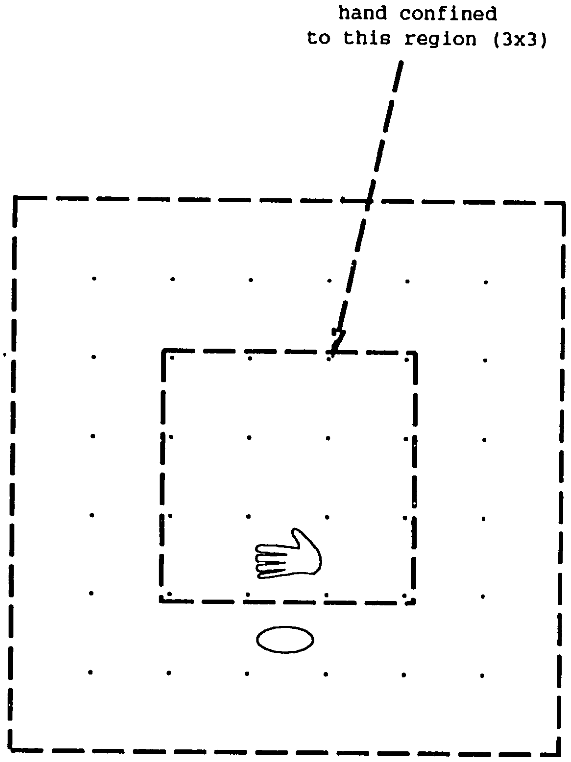

Figure 1-1: The hand can move within a 3x3 area in the area in front of the body.

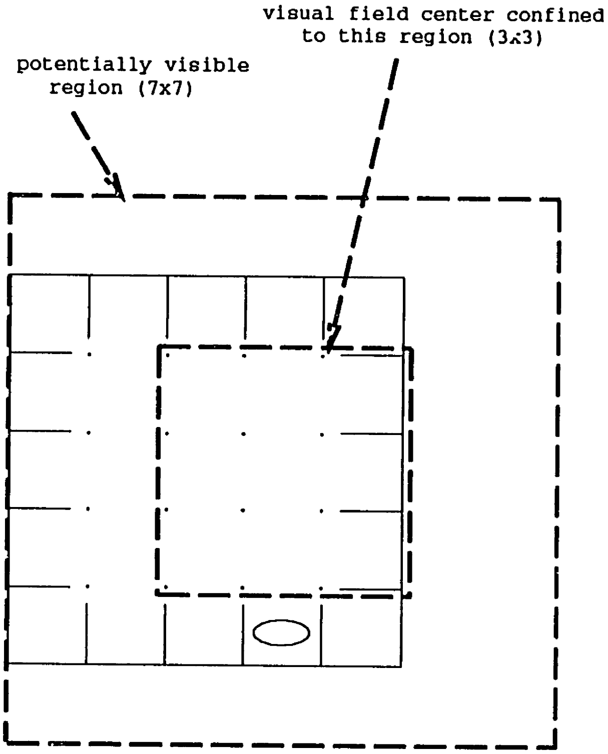

Figure 1-2: The visual field is 5x5 and can assume nine different orientations, for a total visual region of 7x7.

- **handf, handb, handr, handl**: These actions move the hand incrementally forward, backward, right or left.
- **eyef, eyeb, eyer, eyel**: These actions shift the glance orientation incrementally forward, backward, right or left.
- **grasp**: This action closes the hand, grasping any movable object which is immediately to the left of the hand (near its "fingers") unless the hand was already closed. Once closed or grasping an object, the hand remains in that state for three time units, unless explicitly opened in the interim. Moving the hand moves any grasped object.
- **ungrasp**: This action opens the hand, releasing any object that had been grasped.

Table 1.1: The primitive actions

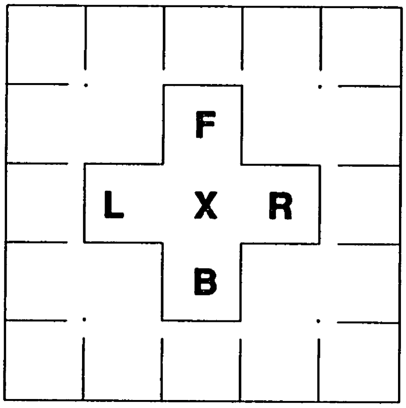

Figure 1-3: Five foveal regions in the center of the visual field (front, back, right, left and center) provide detailed visual information.

to whether or not the hand is closed and (possibly) grasping an object. The primitive items are further explained in table 1.2 [Dre89, adapted from table 3.2, p. 67].

There are two objects in the world, each of which occasionally (at an average of every 200 time units) moves between a series of four contiguous *home positions* in a clockwise direction — see figure 1-4. The right object is out of the range of the hand and therefore cannot be grasped. The left object can, of course, be grasped and moved about. Both objects are often seen by the simulated robot due to their proximity to the body.

- **hp00,...,hp22**: Haptic-proprioceptive (hand position) items, one for each possible hand position, the hand is confined to a 3x3 area (see figure 1-1). Position (0,0) is in the lower left corner of the range; in figure 1-5, the hand appears at position (2,1) which corresponds to item hp21. In figure 1-1 the hand appears at hp10.

- **vp00,...,vp22**: Visual-proprioceptive (visual position) items, one for each possible glance orientation. Coordinate designates the position of the center of the 5x5 visual field, using same conventions as for hand position; in both figure 1-5 and figure 1-2, the glance is oriented at vp01.

- **vf00,...,vf44**: Coarse visual-field items, one for each of 25 glance-relative coordinate positions. Position (0,0) is in the lower left corner of the current visual field; in figure 1-5, the body appears at vf30 while the hand appears at vf42.

- **fovf00,...,fovf33; fovb00-33; fovl00-33; fovr00-33; fovx00-33**: Visual details corresponding to each of five foveal regions: front, back, left, right and center. See figure 1-3. Each has sixteen arbitrary details. In figure 1-5 the left object is in the front foveal region.

- **tactf, tactb, tactr, tactl**: Coarse tactile items, one for each side of the hand: front, back, right, left.

- **bodyf, bodyb, bodyr, bodyl**: Coarse tactile items, one for each side of the body: front, back, right, left.

- **text0,...,text3**: Detailed tactile items, denoting arbitrary textural details of an object touching the left edge (i.e. "fingers") of the hand.

- **taste0,...,taste3**: Detailed taste items, designating arbitrary surface details of an object touching the front edge (i.e. "mouth") of the body/head.

- **hcl**: Hand closed.

- **hgr**: Hand closed and grasping something.

Table 1.2: The primitive items

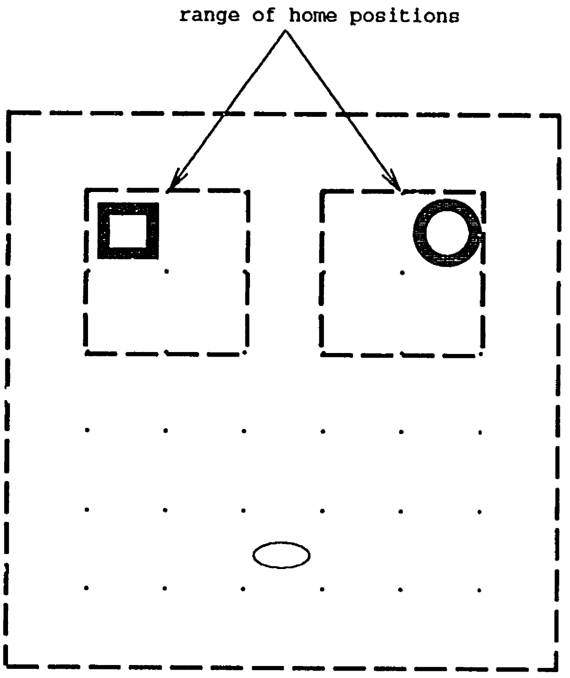

Figure 1-4: The two objects in the microworld circulate in a clockwise direction among a series of four contiguous *home positions.*

The microworld uses three different coordinate systems, microspace, body and glance relative. In each case, the X axis (first position in the coordinate pair) runs left to right while the Y axis (second position) runs bottom to top. This is traditional first quadrant Cartesian coordinate notation [SESA86, p. 92]. Microspace relative coordinates reference the 7x7 world directly where the lower left hand corner is position (0,0) and the lower right hand corner is position (6,0). Body relative coordinates are often used when referring to the center 3x3 area in the microworld. The lower left corner of this area is microspace coordinate (2,2) which is defined as body relative coordinate (0,0). (Translation from microworld to body relative coordinates is accomplished by subtracting 2 from each microworld coordinate; similarly, body relative to microworld coordinate translation is accomplished by adding 2 to each body relative coordinate.) Glance relative coordinates are used when referring to the 5x5 area centered around the current glance orientation. The center of the 5x5 glance relative area is defined as glance relative coordinate (2,2) with the lower left corner of this area defined as glance relative coordinate (0,0).

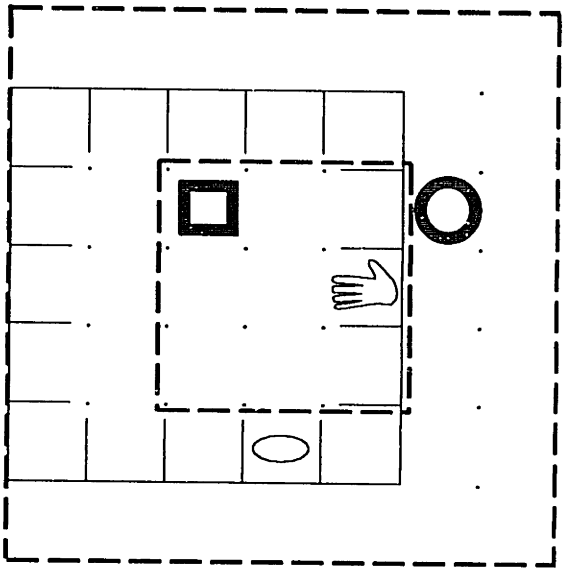

Figure 1-5: A sample microworld situation. The hand and the left object are in view, while the right object is out of view.

In figure 1-5 the glance is oriented at body relative visual position (0,1) and the hand is at body relative hand position (2,1). The body is visible via the coarse visual field items at glance relative position (3,0), the hand at (4,2), and the left object at (2,3). The right object is not visible. The detailed visual information for the left object is present in the front foveal items. If the hand were in body relative hand position (1,2) adjacent to the left object, it could grasp it. The right hand object is currently out of reach. See table 1.2 for more examples using the various coordinate systems.

It is important to note that the names given to each primitive action and item are for purposes of human readability only. The microworld system provides a series of ten functions corresponding to the ten primitive actions and a series of functions which return the status of each of the primitive items. The schema mechanism begins with absolutely *no* knowledge about which actions and items are related to one another.

1.4 The schema mechanism

The schema mechanism, by utilizing the simulated robot in the microworld, attempts to acquire knowledge about its domain through analysis of its experiences. In the appropriate situations, the system can create three different types of structures to embody acquired knowledge: *schemas*, *synthetic items* and *goal-directed actions.*

1.4.1 Schemas

Each schema expresses a specific belief about causality in the microworld and is defined by a context, action and result. The context defines the microworld preconditions under which the schema can be activated. If a schema is activated, its corresponding microworld action is executed. The result indicates those elements of the microworld state which should change when the schema is *activated* (context satisfied and action executed) — in some sense, indicating what the *effects* of the action are when performed in the given context. Essentially, each schema expresses the context-dependent results of a given action.

The context and result can be single items or conjunctions of items, or be empty. For each included item, the context and result indicate if it is positively or negatively included. A context is considered *satisfied* if each included item matches the current state of the microworld — if an item is positively included, it must be on in the current microworld state, and similarly for negative inclusion and off. A schema is considered *applicable* if its context is satisfied and no overriding conditions exist. (Overriding conditions are detected by a schema's *extended context*, see section 2.1.4 for more details.) If a schema is applicable and its action is taken, the schema has been *activated.* The result *obtains* if each item included in the result matches the state of the microworld after taking the action — if a schema is activated and its result obtains, the schema is said to *succeed.* Note that both primitive and synthetic items (discussed later) can be included in a context or result — however, synthetic items can also be in an *unknown* state, which for purposes of satisfying a context or achieving a result does not match positively or negatively included items.

A schema is notated in the form *context/action/result*, where negated items are indicated by a – and conjunctions of items are constructed by placing *&* signs between the items (by convention, the *&* can be omitted in the case of items with single letter names). For example, the schema in figure 1-6 is *p–qr/a/xy.*

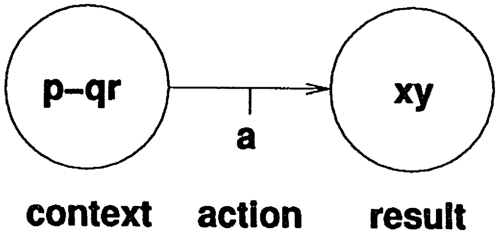

Figure 1-6: A basic schema.

A schema is not a *rule* which indicates that the action should be performed when the context is satisfied; the schema just indicates what would happen *if* the action was performed. Note also that the results indicated by a schema are by no means guaranteed — a reliability measure, which indicates how often the result obtains when the schema is activated, is kept by each schema. Schemas may exist with arbitrarily low reliability — as a particular result does not necessarily follow with regularity, schemas cannot be thought of as rules. The notion of a rule also usually includes the notion that a given action should not be performed unless the preconditions are met. In this learning system, each action can be performed at any time — each schema merely asserts what happens when the action is performed when all context conditions are satisfied. The context therefore should *not* be considered a prerequisite for the performance of the action. It is also possible that items not included in the result will change state — the result is not necessarily exhaustive. Finally, a particular schema says absolutely nothing about what might happen if its action is taken when its context is not satisfied.

1.4.2 Constructing new schemas via marginal attribution

The system begins with a set of ten *bare schemas*, one for each primitive action. A bare schema has an empty context (one with no items), and therefore can be activated at any time. A bare schema also has an empty result, and therefore does not make any prediction whatsoever as to changes in the microworld state due to taking the indicated action (see figure 1-7).

As the system executes, a technique known as *marginal attribution* is used to discover statistically important context and result information. This information is then

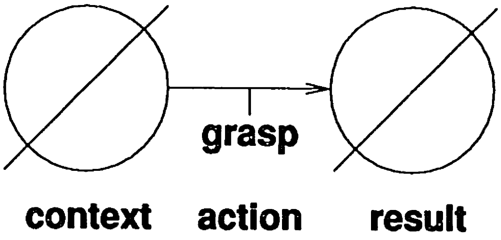

Figure 1-7: A bare schema for the *grasp* action.

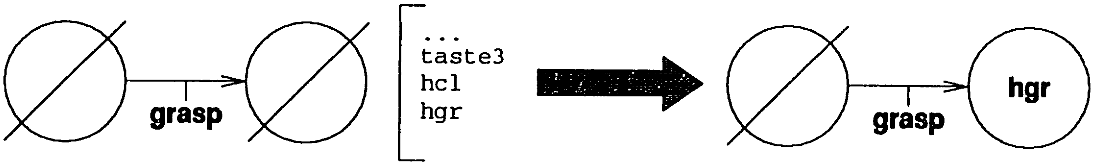

Figure 1-8: The extended result of the bare schema */grasp/* detects that */grasp/hgr* should be spun off.

used to fine-tune existing schemas by creating modified versions of them. Marginal attribution succeeds in greatly reducing the combinatorial problem of discovering reliable schemas from an extremely large search space without prior knowledge of the problem domain.

Result spinoffs

Many different results may occur from the execution of a given action. For every *bare* schema, this facility tries to find result transitions which occur more often with a particular action than without it. For example, my hand ends up closed and grasping an object much more often when the grasp action is taken than with any other action. Results discovered in this fashion are eligible to be included in a *result spinoff* — a new schema identical to its parent, but with the relevant result item included (see figure 1-8). The marginal attribution process can only create result spinoffs from bare schemas.

Specifically, each bare schema has an *extended result* — a structure for holding result correlation information. The extended result for each schema keeps correlation information for each item (primitive or synthetic). The positive-transition correlation is the ratio of the number of occurrences of the item turning on when the schema's action has been taken to the number of occurrences of the item turning on when the

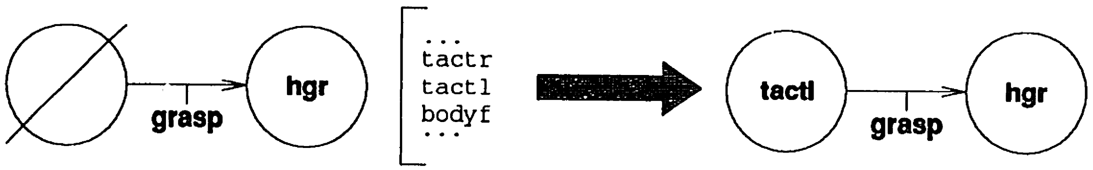

Figure 1-9: The extended context of the schema */grasp/hgr* discovers that *tactl* improves its reliability and therefore *tactl/grasp/hgr* is spun off.

schema's action has not been taken. Similarly, the negative-transition correlation is the ratio of the number of occurrences of the item turning off when the schema's action has been taken to the number of occurrences of the item turning off when the schema's action has not been taken. Note that an item is considered to have *turned on* precisely when the item was off prior to the action and on after the action was performed and similarly for turning off. The correlation statistics are continuously updated by the schema mechanism and weighted towards more recent data. When one of these schemas has a sufficiently high correlation with a particular item, the schema mechanism creates an appropriate result spinoff — a schema with the item positively included in the result if the positive-transition correlation is high, or a schema with the item negatively included in the result if the negative-transition correlation is high. These simple statistics are very good at discovering arbitrarily rare results of actions, especially when the statistics of the non-activated schemas are only updated for *unexplained* transitions. A transition is considered explained if the item in question was included in the result of an activated schema with high reliability (above an arbitrary threshold) and it did, in fact, end up in the predicted state.

Context spinoffs

For schemas which have non-negligible results, the marginal attribution attempts to discover conditions under which the schema obtains its result more reliably. To extend the example, my hand ends up closed and grasping something *much* more often if I can feel an object touching the left edge of my hand before I close my hand with the grasp action. This information is used to create *context spinoffs* — duplicates of the parent schema, but with a new item added to its context (see figure 1-9).

Schemas with non-empty results have an *extended context.* For each item, this structure keeps a ratio of the number of occurrences of the schema succeeding (i.e. its result obtaining) when activated with the item on to the number of occurrences of the schema succeeding when activated with the item off. If the state of a particular item before activation of a given schema does not affect its probability of success, this ratio will stay close to one. However, if having the item on increases the probability of success, the ratio will increase over time. Similarly, if having the item off increases the probability of success, the ratio will decrease. If one of these schemas has a significantly high or low ratio for a particular item, the schema mechanism creates the appropriate context spinoff — a schema with the item positively included in the context if the ratio is high, one with the item negatively included if the ratio is low.

There is an embellishment to the process of identifying context spinoffs. When a context spinoff occurs, the parent schema resets all correlation data in its extended context, and keeps an indication of which item (positively or negatively included) was added to its spinoff child. In the future, when updating the extended context data for the parent schema, if that item is on (if positively included in the spinoff) or off (if negatively included in the spinoff) the trial is ignored and the extended context data is not modified. This embellishment means that the parent schema has correlation data only for those trials where there is no more specific child schema, and it encourages the development of spinoff schemas from more specific schemas rather than general schemas.

Redundancy is also reduced by a further embellishment. If at a particular moment in time, a schema has multiple candidates for a context spinoff, the item which is on least frequently is the one chosen for a context spinoff. The system keeps a *generality* statistic for each item which is merely its rate of being on rather than off — it is this statistic which is used when deciding between multiple spinoff possibilities. This embellishment discourages the development of unnecessary conjunctions when a single specific item suffices [Dre89, p. 104].

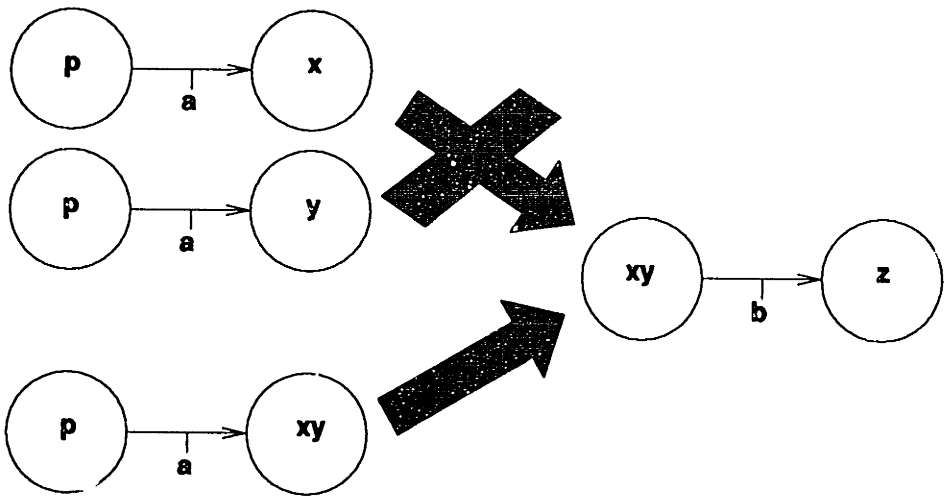

Figure 1-10: Two schemas which predict two items separately can not chain to a schema which has both items in its context: a schema with a conjunctive result is required.

Conjunctive contexts and results

The context can be iteratively modified through a series of context spinoffs to include more and more conjuncts in the context. For a variety of reasons, but primarily to avoid combinatorial explosion, a similar process for result conjunctions is undesirable [Dre89, pp. 105-6]. The marginal attribution process therefore requires that result spinoffs occur only from bare schemas, and only one relevant detail can be detected and used as the result for the spinoff schema. However, conjunctive results are necessary if schemas should be able to *chain* to a schema with a conjunctive context. (The goal-directed action facility in particular depends greatly on the detection of chains of reliable schemas where each schema has a result which satisfies the context of the next schema in the chain. See figure 1-10.) This problem is solved by adding a slot to the extended result of each bare schema for each of the conjunctions of items which appear as the context of a highly reliable schema. Statistics are kept for these in the same fashion as those kept for single items, and if one of these conjunctions is often turned on as the result of taking a given action, a result spinoff occurs which includes the entire conjunction in the result. Effectively, this process is able to produce schemas with conjunctive results precisely when such schemas are necessary for chaining.

Summary of marginal attribution

Schemas created by the marginal attribution process are designed to either encapsulate some newly discovered piece of knowledge about causality in the microworld (result spinoff) or to improve upon the reliability of a previous schema (context spinoff). By continuously creating new versions of previous schemas, the system iteratively improves both the reliability and the scope of its knowledge base. It is interesting to note that once created, a schema is never removed from the system. Rather, it may be supplanted by one or more spinoff schemas which are more *useful* due to higher reliability and greater specifity.

1.4.3 Goal-directed actions

Schemas of arbitrarily high reliability can be thought of as *rules* in that if the context is satisfied, taking the indicated action reliably produces the given result. Therefore, once a number of reliable schemas have been produced, it becomes fairly simple to reach a given goal through planning. Over time, the system becomes able to chain various schemas together to produce a variety of desired results. For any desired result, the mechanism can create a *goal-directed action*, an action which is designed to produce the given result. These new abstract actions give the mechanism an ability to bring about a desired result through a number of intermediate actions, and to treat this process as if it were a single discrete action.

In [Dre89], a goal-directed action is created whenever a particular item or conjunction is highly *accessible* — when, at each clock tick, there is usually some chain of reliable schemas which starts with an activatible schema and ends in a schema with the item or conjunction positively included in the result. [Dre91] creates a goal-directed action whenever a result spinoff has a unique result. The reimplementation uses the method from [Dre89] as it reduces the proliferation of many actions early on, but the method in [Dre91] is simpler, less compute-intensive and seems more cognitively realistic.

When a goal-directed action is created, a bare schema is constructed which has the new goal-directed action and an empty context and result. The marginal attribution algorithm will then attempt to build reliable schemas which utilize the goal-directed action and encapsulate knowledge about the goal-directed action. (For more details about goal-directed actions see [Dre89, section 3.4.2].)

[Dre91] and [Dre89] use *composite action* where I have chosen to use the term *goal-directed action.* An informal discussion group concluded that *composite* is a word overloaded with meaning — in particular, it suggests the treatment of a specific series of actions as a single action as in the mathematical operation of composition where one constructs a new function by defining it as the result of the sequential use of two separate named functions [SESA86, p. 134]. It was therefore proposed to use the term *goal-directed action* instead, which is more precise in meaning, as a goal-directed action will activate whichever series of schemas will most likely achieve the desired goal state — it does not activate the same series of schemas each time it is executed.

1.4.4 Synthetic items

There are certain concepts that the primitive items are unable to express, for example, that a particular object is present at a particular location while the glance orientation is such that the object is out of view. The schema mechanism contains a facility for building *synthetic items* — items which, when on, indicate that a particular unreliable schema, if activated, would succeed. Suppose a schema */move glance orientation to vp01/fovf02* is very reliable if the left hand object in the microworld is in the correct position (see figure 1-5, if the glance orientation is at *vp01* and the left hand object is in the indicated position, it is in the forward foveal region, and could turn *fovf02* on). However, this object spontaneously moves between four different positions and so, on average, is only in the correct position about one-fourth of the time. Notably, this schema, if activated and successful, will continue to be very reliable for some period of time (equal to the duration that the object remains in that position), even though on average it is normally not very reliable. To discover such situations, the schema mechanism keeps a *local consistency* statistic which indicates how often the schema succeeds when its last activation was successful. If a schema is unreliable but highly locally consistent, the mechanism constructs a synthetic item — an item which, when on, indicates that the schema (its *host schema*), if activated, would succeed. Effectively, such an item, when on, predicts what the result of activating the host schema would be. For a variety of reasons (see [Dre91, section 4.2.3]), synthetic items are fundamentally very different from primitive items and express concepts which are inexpressible through any conventional combination of primitive items.

Primitive items get their state directly from the microworld. On the other hand, the schema mechanism itself must maintain and update the state of all synthetic items. The rules the mechanism uses to determine the state of a given synthetic item are summarized in table 1.3. The use of synthetic items effectively allows the schema mechanism to invent new concepts — concepts which are not expressed well by the microworld or cannot be expressed by conjunctions of boolean values at all.

1.4.5 Control

The CM2 implementation cycles between periods where schemas are chosen for activation on the basis of their value, and periods where the system is emphasizing experimentation with recently created schemas [Dre89, section 3.2.2]. The primary goal of the reimplementation of the schema mechanism is to validate the results found in [Dre91] and [Dre89]. There is no analysis of the ability of the CM2 implementation to find and obtain states of high value in either source, rather, the results presented are the structures (schemas, goal-directed actions and synthetic items) which the system built in a reference run. As goal-seeking behavior is not documented in the results of either source, and this reimplementation is an attempt to verify the results, the reimplementation detailed in this thesis does not need to cycle between periods of goal-seeking and experimentation, and therefore doesn't. The reimplementation also does not make use of any notion of *value* (see [Dre89, pp. 78–83] for a discussion of value). Rather, the reimplementation merely selects one of the currently defined actions at random at each time step. As there are bare schemas for each action, and bare schemas are always activatible, each action is always selectable, and so picking

- **Host schema activated**: If the host schema for a synthetic item is activated, the item is turned on if the schema succeeded. If the schema failed, the item is turned off.

- **Host schema overridden**: If the host schema is context overridden, the synthetic item is turned off. For purposes of updating the synthetic item state, a schema is considered overridden if an item is correlated by its extended context at least 75% in the direction opposite the current state of the item. See section 2.1.4 for more details.

- **Context spinoff**: Context spinoff schemas may be created from the host schema in an attempt to improve the reliability of the host schema. These spinoff schemas have the same action and result as the host schema. Whenever a reliable schema is applicable, its parent schema is checked to see if it is a host schema. The fact that a reliable schema with the same action and result is applicable implies that the host schema would succeed if activated, and therefore the synthetic item is turned on. If a reliable schema is applicable, but overridden, its parent schema is not checked.

- **Result predictions**: If a reliable schema which contains a synthetic item in its result is activated, the mechanism assumes that the schema succeeded, and turns the item on (if positively included) or off (if negatively included).

- **Local consistency**: When the mechanism turns a synthetic item on or off for any of the reasons listed here, the item stays in that state for the length of the expected duration for that transition. (The schema mechanism keeps two statistics for each synthetic item: average duration the item stays on once turned on, and a similar statistic for off.) If that period of time ends without the item being turned on or off by the mechanism, the item is placed in the unknown state.

Host schema evidence has the highest priority when determining the state of a synthetic item, as a synthetic item is defined in terms of success or failure of its host schema. Context and result evidence have the next highest priority — if they disagree, the synthetic item is placed in the unknown state. Local consistency evidence has the lowest priority.

Table 1.3: Rules for determining synthetic item state.

from all actions randomly is perfectly acceptable. This also has the nice side effect of ensuring that all actions are exercised roughly equally.

1.4.6 Summary of schema mechanism

Each of these facilities has an important role. While the marginal attribution technique is a powerful and central part of the system, it can only perform induction from what is already known. Goal-directed actions give the system the ability to abstract the details away from a process which is designed to bring about a desired result, while synthetic items allow the system to invent useful and arbitrarily complex concepts. Together, these two abilities enable the system to discover and define concepts and procedures of its own — contributions to the knowledge base which could not be made by marginal attribution.

1.5 Performance

While the marginal attribution algorithm is fairly compute intensive, especially as the number of schemas increases, it is not intractably inefficient. See [Dre89, section 3.3] for a discussion of the architecture of the CM2 implementation (a parallel machine). This implementation completed its reference run in roughly one day of real time. The reimplementation, somewhat simplified but running on a DECStation 5000/120 with 16 megabytes of memory and using Lucid LISP 4.0, completes a reference run in slightly more than two days of real time. The schema mechanism and microworld, while complicated, are not so compute intensive as to make them cognitively implausible.

Chapter 2

Implementation

2.1 Differences between this work and Drescher's

2.1.1 Goal-directed actions

(Note: the term ***goal-directed action*** is used in this work wherever Drescher's works would use the term "composite action". See section 1.4.3.)

While Drescher's work discusses goal-directed actions at some length, they do not seem to be vital for accounting for many of the results he found. In particular, his work does not include statistics for how the system behaves or performs when engaging in goal seeking behavior.

The reimplementation is focused on evaluating the accuracy of the results presented in [Dre91] and [Dre89]. This primary goal, combined with a desire to finish this work in a timely fashion and some technical problems surrounding storage of goal-directed action controller data in reasonable amounts of memory, led to a decision to leave most of the goal-directed action ideas unimplemented.

In particular, the reimplementation uses the method in [Dre89] for determining if a goal-directed action should be created. When a new goal-directed action should be created, the reimplementation merely displays a message to this effect. The reimplementation does *not* create bare schemas for the new goal-directed action, and therefore none of the schemas created by the reimplementation pertain to goal-directed actions. The reimplementation does not create goal-directed action structures, does not update them, and cannot activate them.

In the reimplementation, only positive items or conjunctions (with items which are positively or negatively included) are eligible to be the goal of a goal-directed action. Having a negated conjunction as a goal would be fairly useless, as it is equivalent to a disjunction of negated items, which is something the system doesn't work with or understand. A goal which is the negation of a single primitive item may be useful in some rare cases, but generally, goal-directed actions which turn a given item on are more useful. The reimplementation currently supports only positive single items, but could very easily be modified to support goal-directed actions which have negated items as the goal. In fact, some support for this is already in place, but it seemed fairly unimportant to finish given that the goal-directed action execution code is incomplete.

With the limited implementation of goal-directed actions, some other statistics and structures are no longer necessary. In particular, duration and correlation statistics for schemas are no longer kept — each primitive action has a duration equal to one clock tick and schemas are never activated for their result due to a simplified control mechanism (see section 2.1.3), so neither statistic is needed.

However, many of the structures and code required for finishing the implementation of goal-directed actions are in place in the code, and with an inspired solution to the memory problem noted above, the code could be finished fairly easily.

2.1.2 Value

Drescher's work describes a system whereby various items are given delegated and instrumental *value*, and the schema mechanism, when engaging in goal seeking behavior, tries to reach states which have high value. As there is no analysis of the ability of the CM2 implementation to find and obtain states of high value in either source and no goal seeking behavior in the reimplementation, the notion of value is not necessary for the reimplementation and therefore omitted. With no notion of value, there can be no notion of the *cost* of a schema, and therefore this statistic is not maintained either. (See [Dre89, pp. 78–83].)

2.1.3 Control

Drescher's work indicates that the schema mechanism should cycle between periods of goal seeking behavior and periods of behavior designed to generate more knowledge about the world [Dre89, section 3.2.2]. Goal seeking behavior is brought about primarily through the activation of schemas which have goal-directed actions and therefore encourage the bringing about of a desired result through the explicit activation of a series of schemas. The schema mechanism selects a schema for explicit activation when it is applicable and contains a number of desired results.

As mentioned above, however, the reimplementation does not support the creation or execution of goal-directed actions. Therefore, at each time step, the reimplementation must only choose between the ten primitive actions which are provided by the microworld. In addition, the reimplementation *never* needs to activate a specific schema for its result, due to the lack of goal-seeking behavior and no notion of value, the system can activate any applicable schema at each time step. Therefore, the reimplementation does not select a particular schema for explicit activation. Rather, at each time step, one of the ten primitive actions is selected and executed. As there are bare schemas for each action, and bare schemas are always eligible for activation, each action is always selectable, and so picking from all actions randomly is perfectly acceptable. This also has the nice side effect of ensuring that all actions are exercised roughly equally. All schemas which share the selected action and have a satisfied context are considered *activated.* Drescher's distinction between explicit and implicit activation is unnecessary, as there is no process of selecting a particular schema for explicit activation in this implementation.

2.1.4 Overriding conditions

[Dre89, p. 106] specifies that a schema which is applicable can not be explicitly activated if an overriding context condition occurs. Due to the simplified control system of the reimplementation, this distinction is unnecessary. The control system implemented does not select a schema for explicit activation and therefore the reimplementation does not look for overriding conditions when picking an action to execute.

(Note for future implementors: in a private electronic mail message, Gary Drescher indicated that the CM2 implementation was designed to override a schema when an item is correlated by its extended context at least 50% in the direction opposite the current state of the item. Example: if the extended context of $/a/x$ indicates that the counter for item p is at least halfway to the value which would force creation of a context spinoff with p positively included, and p is negative at the start of this cycle, then $/a/x$ is suppressed — but it is still considered applicable and if action a is taken by another schema, it will be considered implicitly activated for purposes of updating its statistics. On the other hand $/a/x$ cannot be selected for explicit activation, because of the overriding condition.)

2.1.5 Microworld

There are some minor differences in the microworld from that described in [Dre91, section 6.1]. In particular, the two objects are not in precisely the same position, and an arbitrary decision was made to have both objects rotate clockwise among their home positions (this wasn't specified in the original sources).

Hand motions require that the destination square for the hand is empty — if the hand is currently grasping an object, the destination square for the object must empty also. Note, however, that when the hand moves left, the hand ends up occupying the former position of the grasped object, and vice versa for movements to the right. Again, this was not specified in the original sources.

[Dre91, table 6.2] labels the haptic and visual proprioceptive items as having their origin at position (1,1) while the other items using a 2D representation (such as the visual field items) have their origin at (0,0). This is confusing, and in fact [Dre89] uses a (0,0) origin for all 2D items. The microworld items are precisely as described in [Dre91, table 6.2] except that all (1,1) based 2D items have been translated to (0,0) based systems, for example, *hp* and *vp* items now range from (0,0) to (2,2) instead of from (1,1) to (3,3).

In the reimplementation, if the hand is closed, it is automatically opened after three time units pass. This is different than either [Dre91] or [Dre89], and was changed to show that the precise value of the duration is not important.

2.1.6 Schema mechanism

Gary Drescher answered many questions via private electronic mail and phone conversations. Most of the answers served to illuminate material which was already present in his work — these answers were used in writing the explanations of the microworld and schema mechanism in chapter 1.

One detail which was not adequately explained in either [Dre91] or [Dre89] was precisely when, in the order of events, the system is supposed to randomly move objects and open the hand if it's been closed for more than a given clock tick duration. Gary Drescher explained that part of the job of the mechanism is to differentiate between transitions which are caused by the execution of an action and those which are completely external. Therefore, on each cycle, the system selects an action to perform, executes the action, calls the `clock-tick` function (which randomly moves the objects and opens the hand if necessary, as well as incrementing the clock) and then takes the statistics needed by the schema mechanism. The results of calling the `clock-tick` function are considered to be part of the effect of executing the action on this particular occasion — its effects are attributed to the selected action. (As it turns out, the statistics kept by the schema mechanism are fairly immune to this source of noise in the data.)

2.1.7 Reconciling different sources

[Dre91] and [Dre89] contradict one another occasionally. In each case, the reimplementation is based upon [Dre91]. Some examples from the microworld sections of each source:

- in [Dre91] there are 9 possible hand positions in a 3x3 area, while in [Dre89] there are 25 possible hand positions in a 5x5 area
- in [Dre91] there are 9 possible gaze orientations in a 3x3 area, while in [Dre89] there are 25 possible gaze orientations in a 5x5 area
- in [Dre91] when the hand is closed it remains closed for 2 time units, while in [Dre89] it remains closed for 20 time units (note that the reimplementation actually keeps the hand closed for a total of 3 time units, just to show this is a non-critical value, but this value is intentionally similar to that given in [Dre91])

Again, in all cases, the reimplementation is based on the account in [Dre91] except for the creation of goal-directed actions as noted in section 1.4.3 where the reimplementation uses the accessibility method given in [Dre89].

2.1.8 Piagetian influence

As a reimplementation of major portions of Drescher's work and an investigation into the reliability of his results, this work does not directly reference the psychologist Jean Piaget in any major fashion. Piagetian ideas are clearly the driving influence in Drescher's work and the milestones postulated by Piaget are used as a benchmark for the success of the CM2 implementation. For a discussion of the progression postulated by Piaget and some of his theories see [Dre91, chapter 2]. His ideas are the underpinnings of this work and Drescher's works — without his observations and contributions to child psychology, the results found in these works would be much less interesting.

2.2 Design decisions

The reimplementation work began with a number of important goals in mind. Speed of the code was considered to be quite important, as multiple test runs were desired to give statistical validity to the results. The readability and usability of the code was also a significant goal. Finally, the system had to execute on a general-purpose UNIX platform. These parameters led to a number of key decisions:

- The microworld was implemented and tested separately, to make the system more modular and easier to understand.

- A set of macros were developed to increase the speed of the math computations through LISP declarations without making the code in the rest of the reimplementation unreadable (see appendix sections A.3 and A.4).

- As the schema mechanism code is pretty complicated, an effort was made to use data abstractions whenever possible to increase the readability and usability of the code. In each case, an effort was also made to ensure that the use of these data abstractions would not drastically reduce the speed of the program.

- When necessary, tradeoffs were made to save memory at the expense of additional computational overhead. The speed of the program may have been reduced, but from informal benchmarking, it was determined that virtual memory paging with the larger structures was similarly slow. The result is code which can run well on machines with fairly modest amounts of memory.

Given the complexity of the schema mechanism, it was inevitable that the reimplementation code would be computation and time intensive, but the final version of the code is not intractably so. It also, thanks to the constant development of better UNIX platforms, can be used as the basis for future work which is more computation and memory intensive without switching to a different platform.

It is notable that the final version of the code seems to have achieved a reasonable balance between these design goals.

There are some differences, largely omissions, between the reimplementation and the original sources, as noted above in section 2.1. An incomplete implementation of goal-directed actions, no concept of value and the simplified control mechanism are all different than that proposed by Drescher. However, the marginal attribution algorithm and the notion of synthetic items are completely implemented, and these account for the majority of the results reported in [Dre91] and [Dre89].

2.3 Description of reimplementation code

2.3.1 Microworld

The microworld is implemented as a completely separate unit (in its own LISP package) with an interface consisting of 141 functions which can be called to get each primitive item status, 10 functions which are primitive actions (i.e. may change the item status), a `clock-tick` function (occasionally randomizes object position, automatically ungrasps the hand after 3 time units) plus a function which initializes the world and a function which changes the initial random number generator seed.

The microworld supports the definition and initial placement of world objects. Each world object has 16 `t` or `nil` visual characteristics, 4 `t` or `nil` tactile characteristics, and 4 `t` or `nil` taste characteristics. Each object also has a flag indicating if the object is movable or not — only a movable object can be grasped by the hand. Currently, the only immobile objects are the body and hand.

The state of the microworld is kept internally through a handful of special variables. Nothing other than status messages or the values `t` or `nil` are returned by any routine. A primitive item function which returns `t` indicates to the schema mechanism that the item is currently on, while `nil` indicates the item is off.

The primitive actions are implemented as described in section 1.3. Hand and glance motions work if the resulting position is within the designated area. Hand motions further require that the destination square for the hand is empty — if the hand is currently grasping an object, the destination square for the object must be empty also. Note, however, that when the hand moves left, the hand occupies the former position of the grasped object, and vice versa for movements to the right, so only one vacant square is required for hand movements in either the left or right direction regardless of if the hand is grasping an object or not. Grasping only succeeds in picking up an object if the hand is not currently closed and the object to the left of the hand is movable — whenever the hand is closed, it is automatically opened after 3 calls to `clock-tick`.

The `clock-tick` function always checks to see if the grip should be undone, and at random intervals averaging 200 clock ticks moves all of the non-grasped movable objects to their next home position. It also increments the clock and returns the new value. Because of this use of random numbers (and the modified control system which selects between each of the ten primitive actions at random at each time step), the random number generator start is saved, and new random number generator seeds can be created and saved. This facilitates the ability to repeat a given test run.

The microworld can be initialized (with the current random seed) by calling the `init-microworld` routine, which zeros the clock, opens the hand, places the hand at body relative position (1,1), glance orientation at body relative position (1,1), places the body in the customary microworld position (3,1), and puts two generic objects into the world as well (the initial state of the world is shown in figure 2-1).

2.3.2 Schema mechanism

The schema mechanism code (see appendix section A.2) is nearly four thousand lines long. It was therefore essential to keep the code as organized as possible. Data abstraction was the major strategy used throughout to combat complexity. Macros were utilized to implement the data abstractions in a fashion which would keep the time cost at execution minimal. (The use of macros forced simpler datatypes to be defined earlier in the code than more complex datatypes to avoid compiler warnings.)

In addition, with an ambitious program with many complicated algorithms, it is vital to be able to debug sections of the program separately and together. A series of constants and macros are defined at the start of the file. By changing a given

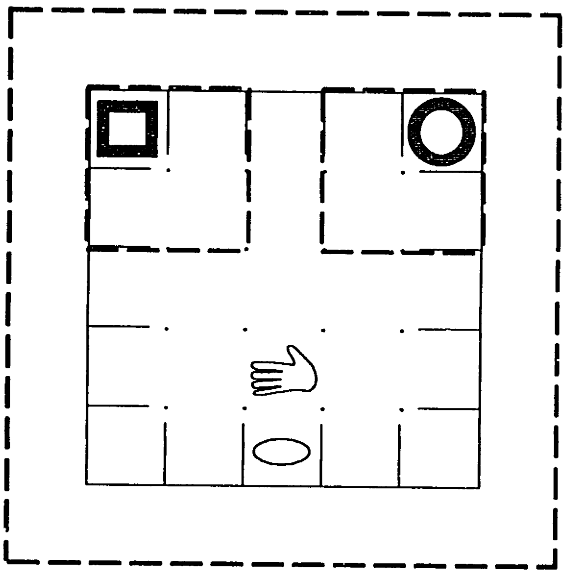

Figure 2-1: The initial state of the microworld.

constant from `nil` to `t` and recompiling the program, execution of the program will send the indicated detailed output to the output file. When a constant is `nil`, all code relating to output for that particular section of the program is eliminated entirely via the macros. This setup gives a great amount of flexibility, as data can be observed as desired without affecting the speed of the program when no output or minimal output is desired.

To aid in the discussion of the code, *simple* datatypes are defined as datatypes which are made up of LISP datatypes and at most one other simple datatype. On the other hand, *compound* datatypes are made up of LISP datatypes and multiple simple datatypes. Each of the major structures of the schema mechanism has a corresponding compound datatype. Each of the compound datatypes also has an array which stores all the created instances of that type.

Simple datatypes

Flags are a simple datatype which can be either true or false. Flag-records and flag-arrays are defined to support storage and manipulation of many flags at the same time. In particular, flag-arrays have some specialized functions which support inclusive ORing of two flag-arrays (`flag-array-ior`) and comparing two flag-arrays to see if all the flags which are true in one flag-array are also true in another flag-

- **flag**: Flags can either be true or false.

- **flag-record**: Flag-records contain 24 flags, each of which can be either true or false. They are used only to help define the flag-array datatype.

- **flag-array**: Flag-arrays contain a number of flags equal to the length of the array times 24. They are used to store large amounts of true/false data in a very compact way.

- **state**: States can be on, off or unknown.

- **state-record**: State-records contain 12 states, each of which can be on, off or unknown. They are only used to help define the state-array datatype.

- **state-array**: State-arrays contain a number of states equal to the length of the array times 12. They are used to store large amounts of state data in a very compact way.

- **counter**: Counters are used to keep extended context and extended result correlation data. They have a toggle flag, a value which ranges from 0 to 15 and a positive flag which indicates the sign for the value.

- **counter-record**: Counter-records contain 4 completely separate counters and are used to help define the counter-array datatype.

- **counter-array**: Counter-arrays contain a number of counters equal to the length of the array times 4. They are used to store large numbers of counters in a compact way.

- **rate**: Rates keep track of how often something occurs.

- **weighted-rate**: Weighted-rates keep track of how often something occurs, weighted towards more recent trials.

- **average**: Averages keep a fixnum value which indicates the average of all the values passed to it via average-update since it was first initialized. They are fairly accurate except for rounding error and are used to keep average durations for synthetic items.

Table 2.1: Simple datatypes defined and used by the schema mechanism code.

array (`flag-array-included-p`). These specialized functions help simplify a number of complicated routines. Flags are implemented as 0 (off) or 1 (on) and flag-records are fixnums which hold 24 flags. Flag-arrays are simply arrays of fixnums. This representation allows 10 fixnums to contain 240 flags, a big memory savings over using the LISP symbols `t` or `nil` (where 240 atoms consume as much space as 240 fixnums in Lucid LISP 4.0) and an implementation which is under a lot more control of the programmer as opposed to using bit vectors. In particular, the specialized functions take advantage of fixnum comparison operations which are much faster than the corresponding bit vector operations, as 24 flags can be compared at once rather than one at a time, and array initialization is faster as well.

States can be on, off or unknown. As with flags, state-records and state-arrays are defined to store multiple states and help simplify complex algorithms. Functions for state-arrays include methods to copy the on or off states in a state-array into a flag-array (`state-array-copy-pos/neg-flag`) and generation of a human-readable string for state-arrays which represent conjunctions of items (`state-array-get-print-name`), as well as also supporting inclusive ORing of two state-arrays (`state-array-ior`) and comparing two state-arrays to see if all the states which are on/off in one state-array are also on/off in the other state-array (`state-array-included-p`). State-arrays are used often, especially to represent conjunctions of items and schema contexts. States are implemented as 2 (on), 1 (off) or 0 (unknown). One trick which is used occasionally is a *both* state (represented as 3) which includes both on and off states, is impervious to inclusive ORing when using `state-array-ior` and matches both on and off states when using `state-array-included-p`. Similar to flags, state-records are represented as fixnums which hold 12 states, and state-arrays are arrays of fixnums. A simple representation would have each state as a fixnum, whereas this method uses 1/12th of the memory and again supports quicker algorithms for specialized operations on state arrays.

Rates and weighted-rates are used to keep track of how often something occurs. Weighed-rates are weighted towards more recent occurrences, while standard rates are not. Averages keep the average value of all the values given to them since they were created. These three datatypes are used occasionally, again largely to simplify sections of code.

The spinoff detection machinery requires a lot of statistics to be kept in the extended context and extended result of each schema. To keep the memory demands reasonable, the counter method detailed in [Dre89, page 109-11] was used. Each counter has a toggle flag, a value which ranges from 0 to 15 and a positive flag which indicates the sign for the value. The toggle flag is used for purposes of alternating between trials which can give positive evidence for a particular correlation and those which can give negative evidence.

The standard way of utilizing counters is to use them to track whenever an event *occurs* and tabulate statistics to see if either a) having a particular item on or off helps the result to obtain (extended context) or b) taking a particular action helps an item to transition from on to off or from off to on (extended result). Counters are explicitly designed to help gather these statistics. The rules for modifying counter values are summarized in table 2.2. An example might help explain these rules. A counter starts at zero, sign positive. On this trial, the event being tabulated occurs. The sign stays the same, the value is incremented to two. On the next trial, the desired transition does not occur, and the value is therefore decremented. As the resulting value cannot be lower than zero, the value is set to zero, and the sign is still positive as zeros are considered positive. On the next trial, the desired transition does not occur again, so the value is decremented again. The value ends up at two but the sign is now negative. Further negative evidence with the desired transition not occurring will continue to increment the value, eventually maximizing the value at 15 with the sign negative. However, a series of positive evidence trials will decrement the value back to zero, switching the sign to positive, and then increment the value, maximizing the value at 15 with the sign positive. Note that this scheme always exerts pressure towards zero due to the fact that the increment (+2) is smaller than the decrement (-3), which protects against maximization of a counter value due to random noise. The toggle is used to either a) alternate between trials with the item on or the item off (extended context) or b) alternate between trials with the schema's action taken

positive	occurred	counter modification
YES	YES	increment
YES	NO	decrement
NO	YES	decrement
NO	NO	increment

- To increment a counter, add two to its value.
- The maximum value is 15.
- To decrement a counter, subtract three from its value.
- If the value for a counter is non-zero and it is decremented, the resulting value can not be lower than zero.
- If the value for a counter is zero and it is decremented, the sign is set negative and the value becomes two (equal to one correlation in the negative direction).
- A value of zero always has the sign set positive.

Table 2.2: Rules for using counters to detect statistically relevant occurrences.

and those with a different action taken (extended result). Alternating between the two types of trials ensures that the counter accurately reflects the significance of the item in question, as it could otherwise be overwhelmed by a series of trials with a particular item on or a series of trials with a given action taken repeatedly.

The simple datatypes are presented here in a distinct order. Flags and states are very simple notions to grasp. Rates, weighted-rates and averages are only slightly more complex. Each of these five datatypes could find uses in other programs. The most complex of all the simple datatypes is the counter datatype, mostly because the extended statistics algorithm which uses counters is difficult to understand and demands particular behavior from the counters.

Compound datatypes

The compound datatypes use a variety of different datatypes, both LISP datatypes and simple datatypes as defined above, to represent structures which the schema

- **schema**: Each schema structure represents a schema which has been discovered by the schema mechanism or is part of the initial repertoire of the mechanism, with all the added data required for various statistics. Schemas are created in an attempt to encapsulate some bit of knowledge about causality in the microworld and as a springboard to further development of more complicated schemas.

- **item**: Each item represents an item which is either primitive and given to the schema mechanism initially, or a synthetic item which is constructed by the schema mechanism itself in response to a schema which is highly locally consistent but unreliable.

- **conj**: Each conj represents a conjunction of items. Conjs are constructed by the schema mechanism when a schema with a conjunctive context becomes highly reliable, and they support the construction of schemas with conjunctive results. No two conjs can exist with the same conjunction of items, the system never creates duplicate conjs.

- **syn-item**: Each syn-item represents a synthetic item, and stores extra information which is not part of the the item datatype. A synthetic item has two associated data structures, an item which stores the standard information for an item, and a syn-item which stores information particular to synthetic items. The state of a synthetic item is not determined by the microworld, rather, it is determined by the schema mechanism itself, and certain specialized statistics are required to support this.

- **action**: Actions are used to connect a human-readable string to each primitive action. (If goal-directed actions were fully implemented, the structure would be similar to that for synthetic items: two data structures defined for each goal-directed action, an action structure which has a string and a function to call to execute the goal-directed action, and a specialized structure which supports everything necessary for the execution of the goal-directed action.)

Table 2.3: Compound datatypes defined and used by the schema mechanism code.

mechanism uses. The most complicated of these is the schema datatype which is used to store blank schemas which the system starts out with as well as all schemas which it creates during execution.

Each schema has a context, an action and a result which define the schema and never change. Schemas are *never* duplicated, each one is unique. A schema claims that if its context is satisfied, taking the indicated action yields a specified result (with a given reliability factor). Each of these three parts are stored within the schema datatype. If the context is empty, the flag context-empty is set true. Otherwise the context for the schema is stored in the context-array, which can represent conjunctions of items as well as single items. To simplify certain parts of the algorithm, if the context is a single item, the context-single flag is set true, and if a conjunction has been created for the context (the schema is very reliable), the context-conj flag is set true and the conjunction number is stashed in the context-item slot for the schema. The result part of the schema is stored somewhat similarly. If the result is empty, the result-empty flag is set true. If the result is a conjunction, the result-conj flag is set true and the result-item slot holds the conjunction number. Otherwise the result-item slot gives the item number for the result, and if the item is negated, the result-negated flag is set true (conjunctions are never negated).

If the action-gd flag for a particular schema is true, the action for the schema is a goal-directed action and the action-item slot contains the index for the goal-directed action. Otherwise, if the action-gd flag is false, the action-item slot contains an index for a standard action. The action slot contains a function which, when executed, performs the action and modifies the state of the microworld. (The action slot is not set for goal-directed actions, due to the incomplete implementation of goal-directed actions, goal-directed actions cannot be selected for execution. Also, this implementation does not select a schema for explicit activation, rather, it chooses between all the available actions at each time step, so this slot is unnecessary. It is provided merely for support for later development of the system.)

Synthetic items can be defined for each schema. For each schema, if the syn-item flag is true, the reifier slot gives the synthetic item index for the synthetic item

toggle	item	result obtains?	counter modification
ON	ON	YES	event occurred, toggle toggle
OFF	OFF	YES	did not occur, toggle toggle
toggle state = item state		NO	toggle toggle

Table 2.4: Rules for extended context statistics.

created for the schema. First-tick is used to store a clock tick time used in updating the on and off-durations for the reifying synthetic item. If the syn-item flag is false, no synthetic item has been created for this particular schema.

The schema mechanism requires a number of statistics to be kept for each schema. Many of these statistics deal with various situations when a schema is activated (context satisfied and action taken). The reliability of a schema indicates how often the result obtains when the schema is activated (i.e. how often the schema succeeds) and is biased towards more recent trials by using the weighted-rate functions.

If the result of a schema is non-empty, its extended-context looks for items which, when ON or OFF before the action is taken, affect the probability of success. The system keeps track of both positive and negative correlations for each individual item via a series of counters, one per item. When a schema is activated (context satisfied and action taken), the counter for each item in its extended context is updated according to table 2.4. For each item, the system uses the counter toggle to alternate between taking statistics with the item on before the action is taken and with it off. In this scheme, the single counter for each item will go maximally positive if having the item on before the action is taken makes the result occur more reliably, and the counter will go maximally negative if having the item off is relevant. (See table 2.2 for an explanation of precisely how the counter values are updated.) A maximized counter indicates that the schema may be chosen as the parent of a spinoff schema.

(If the schema has a goal-directed action, it also keeps a similar set of correlations in its extended-context-post slot but with respect to the value of items after the goal-directed action is executed. These statistics are used to determine items which need to have their state sustained for the duration of the execution of the goal-directed action [Dre91, section 4.1.6, pp. 80–81]. As mentioned earlier, goal-directed actions are not currently executable, and therefore updating of the extended-context-post statistics is not yet implemented, but the structure already exists to support future program development.)

On the other hand, if a schema has an empty result, its extended-result slot looks for item transitions which appear to be caused by the activation of the schema. The system keeps separate positive and negative transition correlations for each item as well as positive transition correlations for conjunctions which represent the contexts of reliable schemas. It alternates between trials with and without the activation of each schema, for each trial, extended statistics for the current schema and item are only taken if the state of the counter toggle for that item matches the activation of the schema. The rules for updating the counters for each schema are explained fully in table 2.5. A counter will become maximally positive if its item undergoes a transition more often when the schema is activated than when it is not. (For more detail, see table 2.2 which explains exactly how counters are updated.)

The extended-context and extended-result counters indicate when a given spinoff schema should be created. The extended-context for a given schema will have a maximally positive or maximally negative counter which indicates which item should be positively or negatively included in a context spinoff schema. The extended-result for a given schema will have a maximally positive counter in either the positive transition or the negative transition statistics which indicates which item should be positively or negatively included in a result spinoff schema.

As mentioned before, schemas are never duplicated. A group of three structures, context-children, result-children and result-conj-children, keep the mechanism from accidentally duplicating an existing schema without having to search through all the schemas checking for duplicates whenever a new spinoff schema is proposed. For example, when a context spinoff occurs, the parent schema notes that the item was spun off (positively or negatively) and makes a similar notation in its new child. No schema descended from either schema will ever attempt to spinoff another context

Result transition statistics for an item are only taken when:

- The item was (off for positive transition statistics, on for negative transition statistics) at the end of the last cycle.
- Each schema which was not activated this cycle only updates statistics for items which were not explained by the activation of a reliable schema (i.e. the transition must not have been predicted).
- If the schema was activated, the counter toggle for the item must be true, and if the schema was not activated, the counter toggle for the item must be false. This forces the statistics to alternate between trials when the schema was activated and those when the schema was not activated.

Positive transition statistics:

item	activated	predicted	positive counter modification
OFF ⇒ ON	YES	don't care	event occurred, toggle toggle
OFF ⇒ ON	NO	NO	did not occur, toggle toggle
OFF ⇒ OFF			toggle toggle

Negative transition statistics:

item	activated	predicted	negative counter modification
ON ⇒ OFF	YES	don't care	event occurred, toggle toggle
ON ⇒ OFF	NO	NO	did not occur, toggle toggle
ON ⇒ ON			toggle toggle

Table 2.5: Rules for extended result statistics.

schema with that item added. (The context-children slot also supports deferral of taking extended context statistics to a more specific child schema. See section 1.4.2.)

To save memory, a data slot holds many important status bits in a compressed form. The applicable, overridden and activated flags are used by the toplevel control code to keep track of the status of the various schemas. Marked is a flag used by the update accessibility routines to keep track of which schemas have already been visited by the algorithm. Succeeded-last is a flag which indicates that the schema succeeded the last time it was activated and is used to help keep local consistency data. Lc-consy (short for local-consistency) is a rate used to keep the probability of successful activation given that the previous activation was successful. If lc-consy is high, the lcly-cons (short for locally-consistent) flag is set true, a synthetic item is created for the schema, the reifier slot of the schema is set so as to point to the new synthetic item, and the schema mechanism begins to keep track of the on and off duration for the synthetic item.

For purposes of data abstraction, a series of macros are defined for each of the flags and rates which are compressed into the data slot. These macros make it appear as if each flag and rate were defined as their own slots by following the standard Common LISP naming conventions for structures. The use of these macros also makes the code much more readable.

The item structure is used to store information about each primitive and synthetic item. The print-name string, the syn-item-p flag, and code if primitive or syn-item-index if synthetic define the item and never change. A synthetic item is indicated by having syn-item-p true. A primitive item has its code slot bound to the appropriate microworld function (see `init-item`) which returns the state for the item. On each clock tick, the state as returned by the code function is placed in the current-state slot, with the old value placed in the last-state slot. (For synthetic items, the mechanism itself determines the state of the item at each time step, and likewise the the current state is placed in the current-state slot and the old value moved to the last-state slot.) Generality is the rate of the item being on rather than off and is used when selecting between more than one possible context spinoff schema. When many context spinoffs are possible for a given schema, the system picks the "most specific" item, which is defined as the one which is *on* less frequently. This is the item with the least generality. Accessibility is the rate of being reachable by a reliable chain of schemas beginning with an applicable schema, highly accessible items have goal-directed actions created with them as the goal. The gd-created-p flag indicates if a goal-directed action has in fact been created with this item as a goal.

Conjunctions of items (primitive or synthetic) are stored in conj structures (short for conjunction). The items included in a given conj are stored in an item-array (a state-array) with a corresponding pair of positive/negative-flag-arrays which have a flag set if that item is included positively/negatively in the item-array. An inclusion-array indicates which other conjs are included by a particular conj. Example: the conj *a & b & c* would include the conjs *a & b* and *a & c* in addition to others. The flag arrays and the inclusion-array are used to speed up certain algorithms required by the system. Highly accessible conjunctions are eligible to be the goal of a goal-directed action, the data slot for each conj keeps appropriate statistics to support this. Finally, the state of each conjunction is computed at each clock tick and placed in the current-state slot, while the old value is moved to the last-state slot.

Synthetic items are used to designate validity conditions of unreliable schemas which are locally consistent. The unreliable schema which causes creation is called the host-schema, while the synthetic item is the schemas "reifier". Synthetic items are included as items in the item array and are treated 100% as if they were primitive items, they can be in the context and result of schemas and conjunctions. However, the state of a synthetic item is determined by the schema mechanism rather than by a call to a microworld function. Certain extra data is needed to support the mechanism determining the state directly. This data is stored in a syn-item structure. The host-schema slot contains the index into the schema array for the host schema. Item-index is the index into the item array for the entry for this synthetic item. Maybe-state is used by the routines which calculate the state for the synthetic items — it is merely a place to stash an intermediate value before deciding what the current value is. On-duration and off-duration are the lengths of time the synthetic item tends to stay on or off once placed in that state respectively. Set-time is the clock tick when the item was last modified, while unknown-time is the clock tick when the item should be automatically set unknown.

The action datatype is simply a way of relating human readable print names with a series of functions which are called to execute a given action. It isn't even a LISP structure, rather, it is a pair of arrays which have the same relative indexing.

The rest of the code

It is notable that even with the complexity of the algorithms required for the schema mechanism that of approximately 3800 lines of code, roughly 1800 lines are used to implement the datatypes referenced above. A solid grip of all of the datatypes is essential before the remainder of the implementation code makes sense. The remaining code is precisely what would be expected based on the description of the algorithms in chapter 1 and the datatypes as described above and should be fairly understandable. Some comments on some of the more unusual and complex functions are in table 2.6.

The main function **run** ties the whole system together. Everything is initialized, and a loop is entered which saves the output from each 50 clock ticks into a different file. For each clock tick, the system first shows the state of the microworld. Applicable schemas are marked, accessibility of items and conjunctions is updated and goal-directed actions are created. (In the current implementation, goal-directed actions aren't fully supported, and so the system merely outputs a message indicating that the given goal-directed action would have been created.) An action is selected at random and executed, and the microworld `clock-tick` function is called. The state of all items and conjunctions is then updated. Schemas which were applicable and shared the executed action are then marked activated. The state of the synthetic items is then updated. Each schema is marked to indicate if its result obtained, and this data is used to update the reliability factor for each schema. Predicted results are noted, and this information is used in updating the extended statistics for each schema. Note that the extended statistics are taken *after* the microworld `clock-tick`

- **schema-update-print-name**: This function takes a schema which has a context, action and result defined as described in the schema datatype code and in this document and sets its print-name slot to a human-readable description of the schema. This function is somewhat sensitive and will *break* or produce unusual results if the flags referenced in the schema datatype code are not set correctly.

- **init-everything**: An unusual thing about this function is the selector argument, which selects between three different microworlds. 0 selects the standard microworld, 1 selects a microworld with only the left hand microworld object and 2 selects a microworld with only the right hand object. This supports more than one type of test run. This function also takes a random state filename as an argument which allows the system to begin from a different random number generator seed for different test runs. The main `run` function and the batch files for the test runs take advantage of these two arguments.

- **update-accessibility**: This function is extremely complicated and requires a lot of convoluted computations. It is also heavily commented. Note that [Dre91] does not use this method to decide when to make a goal-directed action. [Dre91] uses a simpler method which just creates a goal-directed action for each item or conjunction which appears as the result of a schema. See section 1.4.3. The method implemented here finds items which are accessible through a path of .75 reliability forward from any currently applicable schema. The maximum length of a path is 5 schemas and the reliability of a path is the product of the reliabilities of each schema on the path. Also, only positive accessibility is important, as the program is finding items and conjunctions which it wants to make composite actions for, and negative conjunctions don't make sense (they would be equivalent to disjunctions), while negative items aren't very interesting as goals (and also would proliferate much too easily).

- **syn-item-update-state**: The functions which implement the synthetic item state algorithm are fairly complicated, but a good grasp of the datatypes and the material in table 1.3 should serve to illuminate the code.

- **run**: As mentioned above in `init-everything`, this function takes a selector and a random state filename as an argument, which it passes on to the `init-everything` function. It also takes a string prefix which is concatenated with a continuously increasing three digit number to produce a series of output filenames for each fifty clock ticks.

Table 2.6: Notes on some of the functions in schema.lisp

function has been called, they are expected to discern between the effects of the action and random effects caused by the `clock-tick` function. Finally the system may add one new schema, any number of conjunctions and any number of new synthetic items at the end of each clock tick.

Chapter 3

Results

3.1 Test runs

[Dre91] and [Dre89] each have results which are drawn from a single *reference run.* For this work, the reimplementation code was executed twenty times with a different random number seed each time in a desire to get more statistically significant proof of the ability of the schema mechanism to acquire large amounts of knowledge about the microworld. The various test runs also serve to demonstrate that certain categories of knowledge seem to form regardless of the order in which various actions are performed.

Each test run was ran until 10000 clock ticks went by, or the space for schemas (3600 schemas maximum) or conjunctions (200 maximum) was exhausted. Many of the test runs ended before clock tick 10000. However, all made it to at least clock tick 7000 before terminating due to saturation and most made it much farther than that. Two test runs made it to clock tick 10000, eight were terminated when they ran out of conjunction space, and ten ran out of schema space. (See the first few lines in tables 3.3, 3.4 and 3.5 for the last clock tick that data was saved for each test run and the reason for early termination if the test run ended before clock tick 10000.)

The reference runs in [Dre91] and [Dre89] terminate when all of the available memory is exhausted. The CM2 implementations also utilize a number of separate processors in parallel, where each processor designates a schema or goal-directed action. [Dre89, p. 94] indicates that the reference run from this source was limited to a total of roughly 3600 schemas due to hardware, while [Dre91, p. 105–6] indicates that the reference run from this source was limited to roughly 7400 schemas.

When using a single UNIX workstation, there is the option of using virtual memory and there are no complications involving dividing up structures among physical processors. Given enough time and patience, and a big enough hard disk for memory paging, substantially more than 7400 schemas can be supported by the reimplementation code. The lower number of 3600 schemas was chosen and used due to the desire to perform a series of test runs, especially as the running time for each clock tick in a single processor serial implementation increases linearly with the number of schemas. It was deemed unreasonable to spend more than the forty CPU-days required to execute the twenty test runs.

3.1.1 Output

The output file from each test run lists the schemas, synthetic items and goal-directed actions which were built and the clock tick when they were built. The program also checks through all of the schemas every 50 clock ticks, and outputs the number of each schema which is highly reliable. An abridged output from a reference run is in appendix section B.1. It has the reliability data removed and is only the output from the first 2300 clock ticks. When the reliability data is included and the test run complete, the output from a test run is a formidable amount of information to wade through and understand, as it can run over one megabyte in length. On the other hand, with one line per structure built by the program and a space-saving tradeoff for reliability of schemas (just outputting the schema numbers of those above .9 reliability each 50 clock ticks, as opposed to outputting some continuous reliability measure or outputting the actual reliability numbers for all schemas each 50 clock ticks), this output is basically as minimal as it can possibly be and still summarize most of the important things which happened during a given test run.

3.1.2 Analysis program

An analysis program was written which reads in an output file from a test run and analyzes it, selecting certain schemas, categorizing them and calculating certain statistics for each category. By using this program, all twenty test runs were held up to a consistent amount of scrutiny within a constrained amount of time. Another benefit of using the analysis program was that certain regularities among the various test runs were made obvious, many of which would have probably been obscured if the raw output files had been analyzed by hand. The code for this program is in appendix section A.5, and sample output from the program in in appendix section B.2.

Of course, this approach brings along with it the potential for abuse, as it is possible that the program itself is obscuring schemas which are important by not outputting schemas which do not fit a category and are not highly reliable. As used in this work, the analysis program has been extremely useful and there have been no perceived negative effects. Indeed, the major purpose of the analysis program is to establish a focus on the schemas which belong to various categories or are highly reliable and therefore worthy of further investigation.

All analysis of the output file did not concern structures which contained goal-directed actions or synthetic items, as goal-directed actions weren't fully implemented in the reimplementation, and the synthetic items mentioned in Drescher's results reference a goal-directed action. (See 3.2.13.)

3.2 Drescher's results and schema categories

In the result sections of [Dre91], Drescher identifies a number of schemas which he feels are notable and display a significant amount of learning on the part of the schema mechanism. The analysis program discussed above in section 3.1.2 indentifies schemas which belong to these categories, among other things. The following sections summarize these categories and the results found by the original CM2 implementation. A more detailed treatment of this material can be found in [Dre91, pp. 119–141]. Note that some of the examples below are taken directly from Drescher's work, and therefore may reference coordinates which only make sense in the CM2 implementation.

3.2.1 Initial schemas

Both the CM2 implementation and the reimplementation start out initially with ten bare schemas, one for each primitive action. These schemas are */handf/*, */handb/*, */handr/*, */handl/*, */eyef/*, */eyeb/*, */eyer/*, */eyel/*, */grasp/* and */ungrasp/*.

3.2.2 Grasping schemas

The first schema built by the CM2 implementation is */grasp/hcl*. This schema asserts that taking the grasp action results in the hand being closed. This schema is reliable, despite an empty context, as the result follows unconditionally from the action. The marginal attribution process builds this schema quickly because the result occurs only when the *grasp* action is taken. A similar schema, */grasp/hgr*, indicates that the grasp action often leads to grasping an object. However, this action is unreliable, and the CM2 implementation eventually builds *tactl/grasp/hgr* which indicates the importance of having an object to the left of the fingers. A schema is classified as a grasping schema by the analysis program if it has either *grasp* or *ungrasp* as its action.

3.2.3 Coarse visual field shifting schemas

A series of schemas built by the CM2 implementation connect the various coarse visual items to one another via the incremental gaze actions. A typical schema is *vf04/eyel/vf14*. The full complement of these schemas indicate that if an object is seen at a particular position, shifting the gaze will result in that object appearing at a specific adjacent position (depending on which direction the glance action was taken in). Note that the two coarse visual items must relate correctly to one another, for example, moving the glance to the left causes a given object to shift to the right. (See figure 3-1.) There are a total of eighty such schemas, twenty for each glance action,

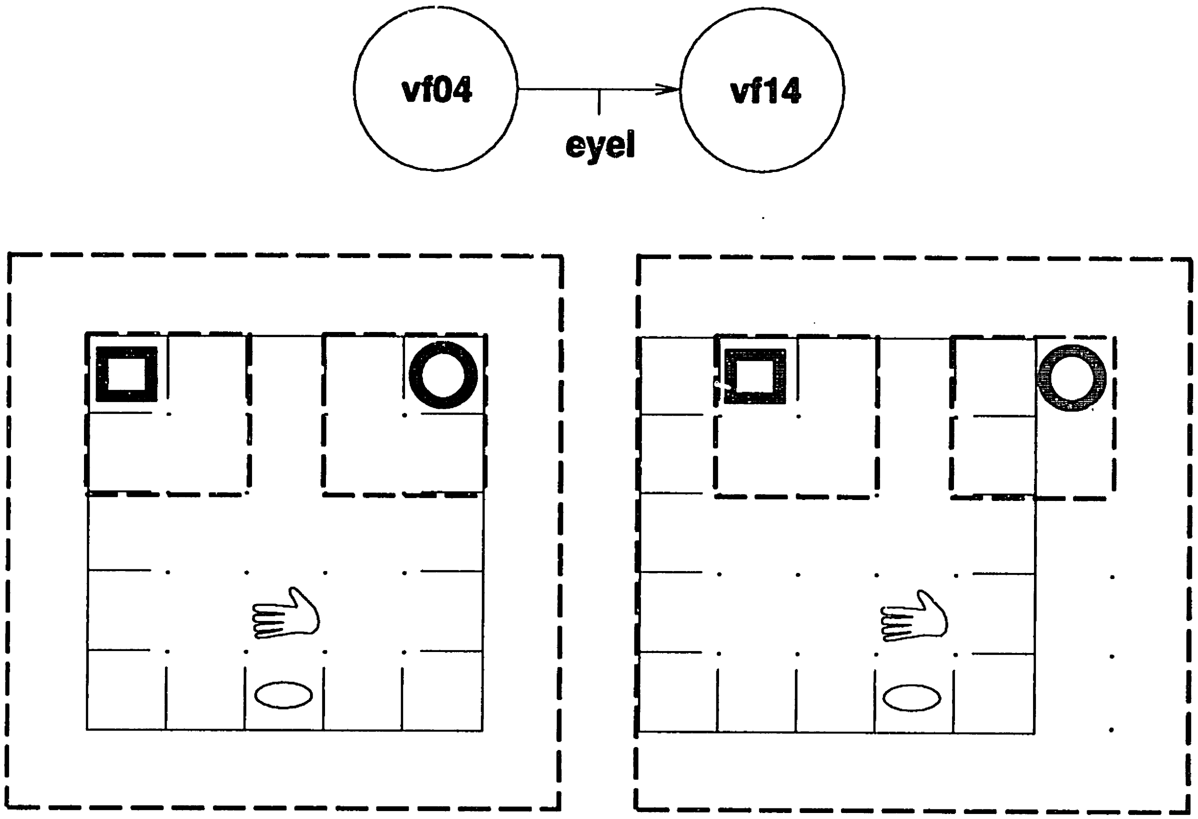

Figure 3-1: Shifting the gaze often causes an object which is within the visual field to appear at a new coarse visual item location. For example, the left hand object in this figure is moved from *vf04* to *vf14* when the glance is shifted to the left.

which together form a network which elaborates the spatial relationships between the various elements of the coarse visual field. The CM2 implementation builds fifty–five of these schemas in its reference run [Dre91, p. 123].

The CM2 implementation also builds twenty–four schemas that concern the special case of moving the image of the body. A typical schema of this type is *vp11/eyel/vf30* where the image of the body appears at *vf20* when the glance orientation is *vp11.* (Again, see figure 3-1, the image of the body shifts from *vf20* to *vf30* through the gaze action, but it is just as valid to use *vp11* as an appropriate context.)

The analysis program catagorizes any schema with the format *vf??/eye?/vf??* (using the traditional UNIX notation where "?" indicates a position occupied by any single character and "*" is used to represent any number of characters concatenated together, including zero characters) as a coarse visual shift schema provided that the two items and action relate correctly to one another as discussed above, and the two items are not identical. It categorizes any schema with the format *vp??/eye?/vf??* as a schema which sees the body via coarse visual items provided that the image of the body when the gaze is oriented at *vp??* appears at a position which relates correctly to the action and the coarse visual item.

The coarse visual shifting schemas are reliable except when either a) the gaze is already at the extreme orientation for the gaze shifting action or b) the object happens to move as a result of a call to the `clock-tick` function after the action has been taken and before the statistics are updated.

3.2.4 Visual field shift limit schemas

The coarse visual shift schemas referenced above are unreliable when the gaze is already in an extreme orientation. In response, the mechanism builds a few schemas like *-vp01&vf04/eyel/vf14* which relate the importance of not having the glance orientation at an extreme position. For example, in figure 3-1, moving the gaze to the left works to shift the object from *vf04* to *vf14*. However, if the action *eyel* was taken again, none of the coarse visual items would shift at all because the glance is already oriented maximally to the left at *vp01*. The schema *-vp01&vf04/eyel/vf14* and others like it are built in response to this inconsistency. There are a very large number of these schemas, the CM2 implementation just begins to learn about them in the reference run [Dre91, p. 123]. The analysis program categorizes a schema as one of this type if it has the form *-vp??&vf??/eye?/vf??*, the coarse visual items relate correctly to the incremental glance action and are not identical, and the visual position item represents an extreme where taking the glance action would not change the gaze orientation.

3.2.5 Foveal region shift schemas

In a similar fashion to the coarse visual field shifting schemas, the mechanism discovers a number of schemas which deal with the shifting of objects within the detailed foveal regions. *fovb11/eyeb/fovx12* is a typical schema which is built by the mech-

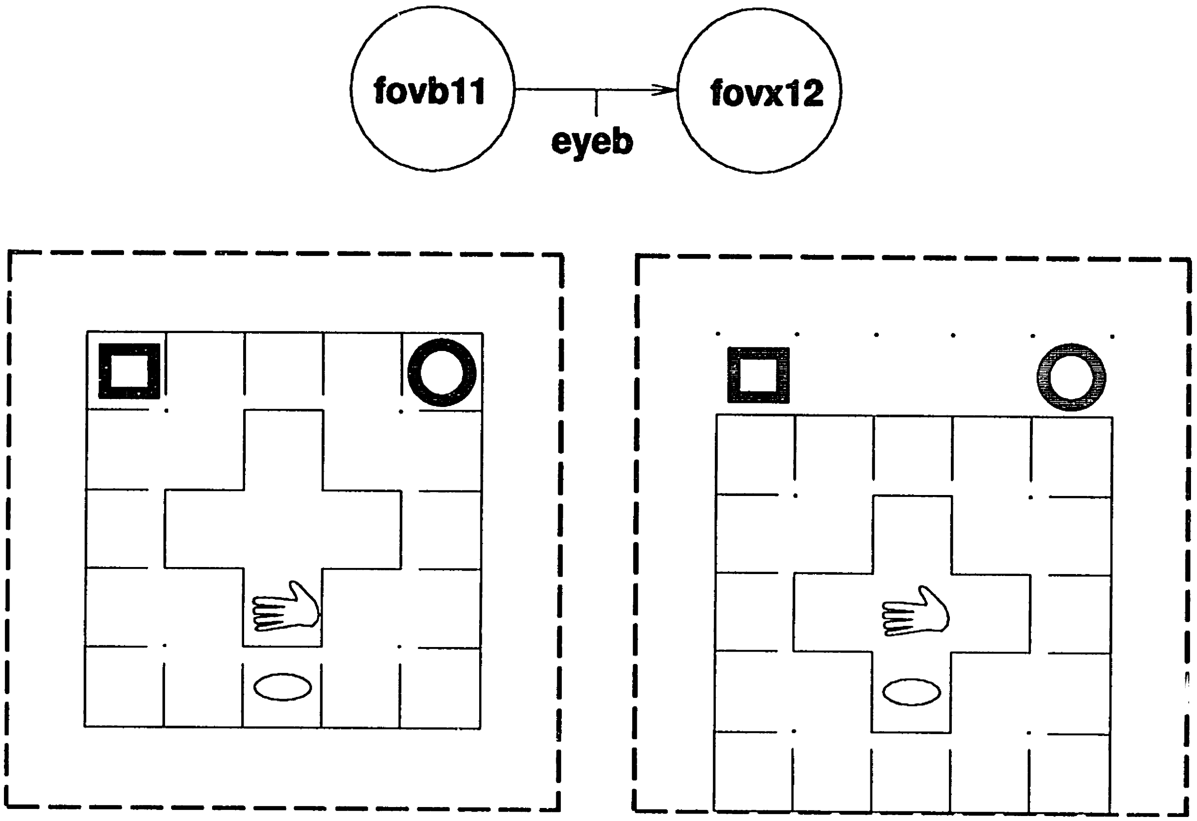

Figure 3-2: The detailed visual appearance of an object often shifts from one foveal region to another when a gaze action is taken. The hand in this figure is shifted from the rear foveal region to the center foveal region when the gaze is shifted to the rear.

anism. Many visual details tend to co-occur between objects, and therefore this schema is just as valid as the more obvious *fovb12/eyeb/fovx12.* Of course, if the mechanism later sees an object which doesn't have both detail 11 and detail 12, the reliability of the first schema may drop, and the second more obvious schema may be built. The analysis program classifies schemas as this type if they have the format *fov???/eye?/fov???* and that the two different foveal regions relate correctly to one another through the indicated gaze action. (See figure 3-2.)

3.2.6 Detail shift schemas

The mechanism also discovers the relationship between the coarse visual field items and the detailed foveal regions. In particular, the system builds a number of schemas such as *vf21/eyeb/fovx12.* These schemas relate a particular coarse visual field position with an eye movement which brings the visual details of the object to a particular foveal region. Note that since *fovb11* is never on unless *vf21* is on, the mechanism will never create *fovb11/eyeb/fovx12* once it has created *vf21/eyeb/fovx12* due to the deferral to a more specific schema discussed in section 1.4.2.

A schema is considered to be a detail shift schema by the analysis program if it has the format *vf??/eye?/fov???* or the format *vf??&fov???/eye?/fov???*. The coarse visual item (and the initial foveal region, if the schema is of the second format) must also relate correctly with the given eye movement and final foveal region.

Figure 3-2 illustrates a number of schemas of this type. *vf21/eyeb/fovx12* indicates that shifting the glance to the rear moves the hand into the center foveal region. *vf20/eyeb/fovb03* is similar, but the body is initially only visible in the coarse visual region, as opposed to the previous example where the details of the hand were already initially apparent in the *fovb* items. Many of the schemas of this category indicate how to shift the gaze so as to make the details of a given object apparent. Finally, a schema like *vf21&fovb11/eyeb/fovx12* may be formed by the mechanism if *vf21/eyeb/fovx12* has been created and the schema mechanism sees sees some objects which do not have visual detail 11 or 12 moving from *vf21* into the center foveal region. These schemas improve on the reliability of their predecessors by constraining which objects have *fovx12* on when moved to the center foveal region.

3.2.7 Visual network schemas

Incremental shifting of the glance changes the glance orientation in a specific and reliable way. *vp11/eyeb/vp10* is a typical schema built by the system in response to this regularity. Figure 3-2 illustrates an example of a successful activation of this schema. These schemas form a lattice which shows the spatial relationship between each of the visual proprioceptive items.

The system also builds schemas such as *vf20/eyeb/vp10* because of the continuous relative position of the body; the body appears at *vf20* when *vp11* is on. The schema *vf20&vp11/eyeb/vp10*, which explicitly relates the two visual proprioceptive items, is built later by the mechanism. Figure 3-2 shows a situation where both of these schemas are also successfully activated.

The CM2 implementation reference run builds 17 of the 24 *vp??/eye?/vp??* schemas, and seven pairs of *vf??/eye?/vp??* and corresponding *vf??&vp??/eye?/vp??* schemas. Note that the CM2 results do not include schemas where the *vp* items did not change. The analysis program recognizes schemas of all three formats as belonging to this category provided that the various items relate correctly to one another and to the gaze action taken, but also accepts schemas where the *vp* items are identical, provided that the gaze is in an extreme position where taking the indicated gaze action does not change the gaze orientation. This increases the number of possible *vp??/eye?/vp??* schemas to 36, and also increases the number of possible pairs placed in this category. These schemas are included in this category because they encapsulate valuable information about the limits of the possible gaze orientations.

3.2.8 Hand movement network schemas

Similarly, the mechanism builds a series of schemas which relate the various haptic proprioceptive items to one another. *hp10/handl/hp00* is a typical schema. The system may build schemas like *taste0/handl/hp00* instead in situations where the hand is immediately in front of the body before taking the hand action. The *taste0* item always occurs with *hp10* as *taste0* is a detail indicating what the hand "tastes like" when in front of the body, and the system makes an arbitrary decision as to which of the two items is included in the context spinoff. Figure 3-3 illustrates both of these schemas.

[Dre91, p. 127] does not clearly indicate if schemas like *hp00/handl/hp00*, where the hand is at an extreme orientation and doesn't actually move, are to be considered part of this network. If they are included, there are a total of 36 schemas, 9 for each incremental hand action. The analysis program includes these schemas in the hand network category; all schemas of the format *hp??/hand?/hp??* are included provided that the hand position items relate correctly to one another and the incremental hand action. If the items are identical, they must indicate an extreme position where the given hand action does not change the position of the hand.

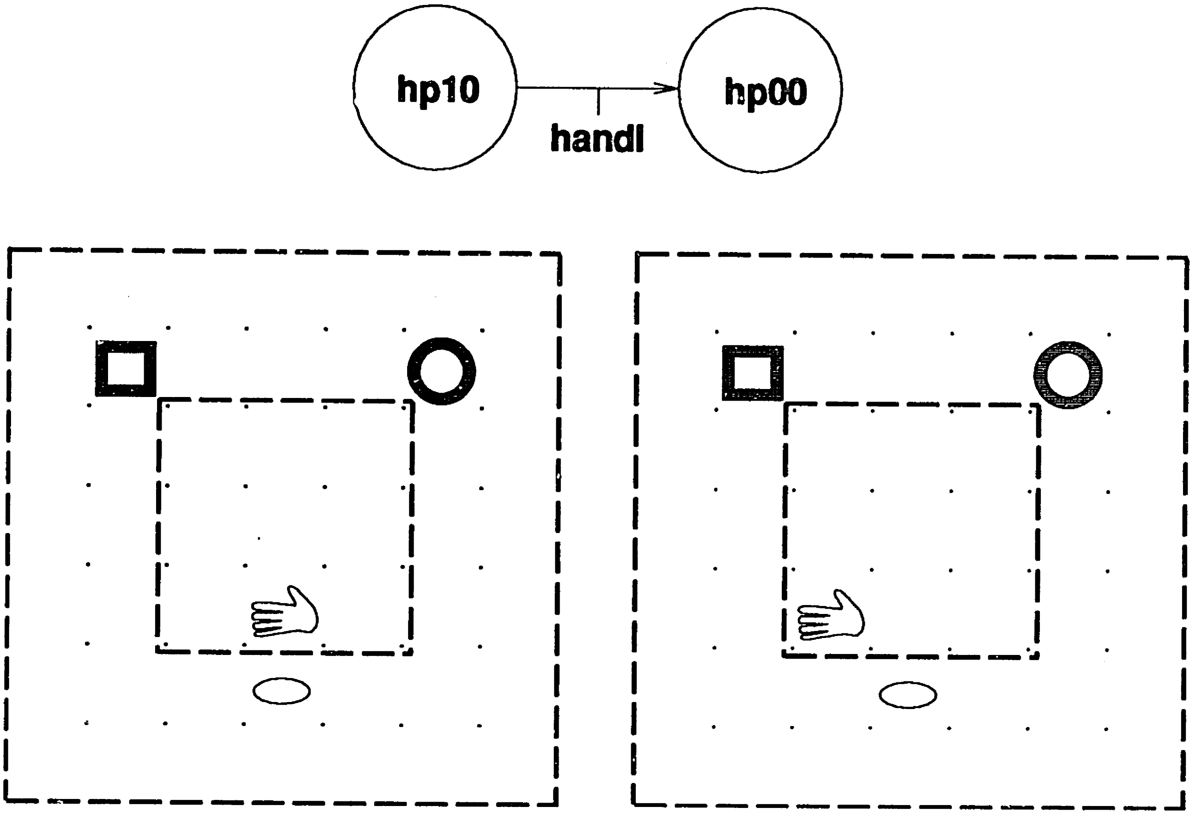

Figure 3-3: Shifting the hand results in the hand moving to a new position.

The CM2 implementation builds all of the schemas in the haptic proprioceptive network in its reference run, with the exception noted earlier where two schemas of the form *taste?/hand?/hp??* are built instead of the corresponding *hp??/hand?/hp??* schema. The analysis program considers these schemas to be part of this classification, as well as *bodyf/hand?/hp??* and *tactb/hand?/hp??*, provided that the hand motion and resulting hp item indicate that the hand was, in fact, immediately in front of the body at *hp10* before the action was taken.

3.2.9 Negative consequence schemas

Taking a glance action or hand motion action changes the current position of the hand or gaze orientation (unless the hand or gaze is already at an extreme position). [Dre91, pp. 126-7] shows */eyel/-vf23*, */eyel/-vp33* and */handb/-hp12* as representative schemas of this type. However, [Dre89, pp. 138-9] shows the schemas *vf23/eyef/-vf23*, *vp12/eyel/-vp12* and *hp34/handb/-hp34*, which is actually a fairly radical difference in definition. The analysis program accepts a schema as a member of this category if it has a *hand* or *eye* action and has a result which is the same as its context, but negated. In other words, the analysis program is looking for schemas of the type described in [Dre89] for this category.

3.2.10 Hand to body schemas

The schema mechanism also learns that when the hand is near the body, it can be brought immediately in front of the body, which affects the taste and coarse tactile sensations felt by the body and hand. Schemas such as *hp11/handb/taste2*, *hp11/handb/bodyf* and *hp11/handb/tactb* are formed by the CM2 implementation to represent this knowledge. These schemas are similar to those which are in the hand network, just as the hand position in front of the body can be referenced to the hand positions around it through hand motions, the inverse can be done as well. The analysis program accepts any schema with the format *hp??/hand?/(taste? or tactb or bodyf)* as a member of this classification provided that the initial hand position and incremental hand action result in the hand being in front of the body at *hp10.*

3.2.11 Seeing hand movements via coarse visual items

When the hand is moved and is currently visible, the transitions taken by the various coarse visual items lead to the formation of schemas such as *vf21/handf/vf22.* (See figure 3-4.) These schemas are unreliable, as the object appearing at a given coarse visual field position doesn't necessarily have to be the hand. The analysis program considers schemas of the format *vf??/hand?/vf??* to be members of this class provided that the two coarse visual items are not identical and they relate to one another and the incremental hand action correctly.

3.2.12 Seeing hand movements via detailed visual items

When the hand starts out in one of the foveal regions, however, the schema mechanism can easily discern between the hand and other objects. A large number of schemas

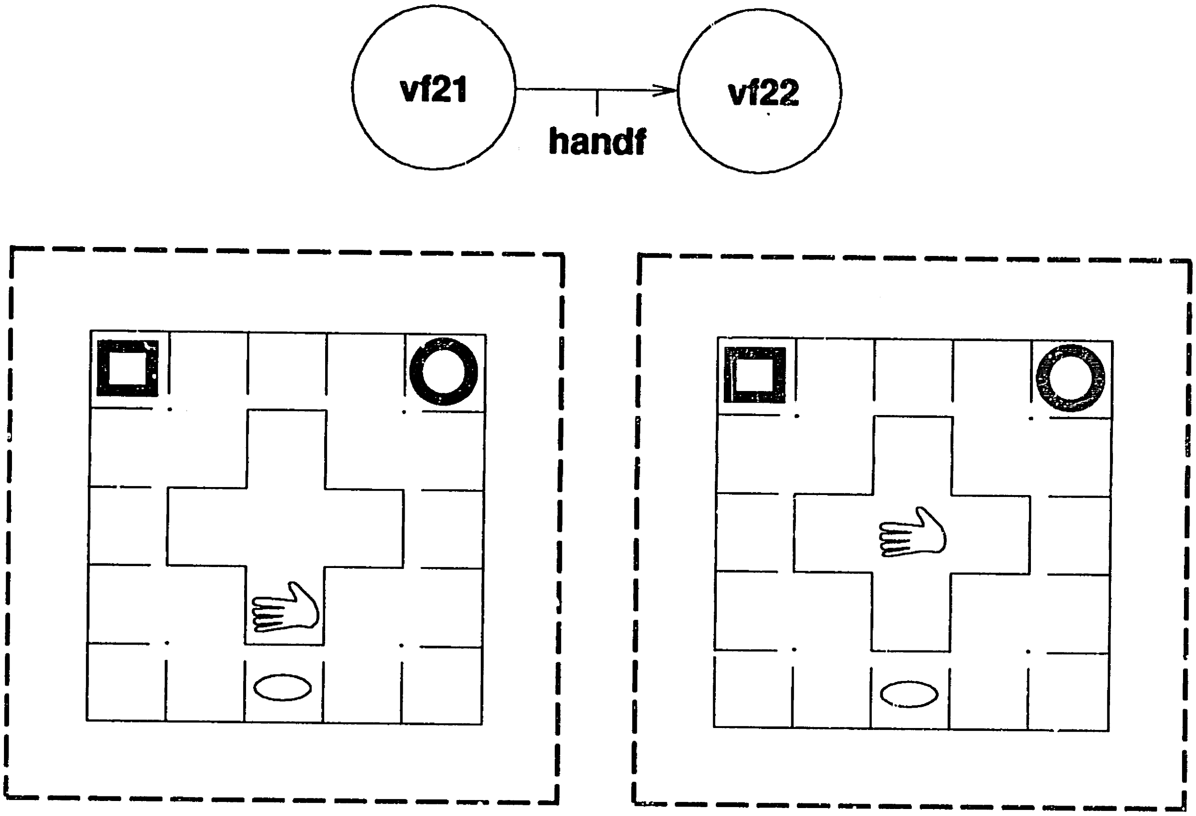

Figure 3-4: Shifting the hand when it is visible often causes a transition to a new coarse visual item (and possibly to a new foveal region).

such as *fovb22/handf/fovx10* form to relate two foveal regions to one another through a hand movement. Over time, the CM2 implementation adds details which allow the mechanism to discern between the hand and other objects which appear in the various foveal regions to the context. This process eventually culminates in the formation of schemas like *SeeHand@21/handf/SeeHand@22* where *SeeHand@21* is shorthand for a series of visual details which serves to distinguish the hand from all other objects. (Position 21 corresponds to the rear foveal region and the fovb visual detail items, while position 22 corresponds to the center foveal region and fovx visual detail items.) The analysis program classifies schemas of the format *fov???*/hand?/fov???** in this category provided that the foveal regions referenced by the first item in the context and result relate correctly to one another and the hand action.

3.2.13 Further results of the CM2 implementation

Drescher's CM2 implementation accomplishes a number of other interesting milestones in its reference run, which are summarized in tables 3.1 and 3.2. Again, each example in the table is taken directly from Drescher's work, and therefore may reference coordinates which only make sense in the CM2 implementation. By a notational convention, if an item or conjunction of items appears in the action position for a schema, the schema has a goal-directed action with the goal of turning each of those items on (if positively included) or off (if negatively included).

The reimplementation code is unable to verify any of these further results, as each of them requires executable goal-directed actions in order to form.

3.3 Results confirming Drescher's work

The results of analyzing the twenty test runs for the schemas discovered and categorized by Drescher are shown in tables 3.3, 3.4 and 3.5. This data is further summarized in table 3.6. Discussion of this data and conclusions can be found in chapter 4. Note that these tables only concern the categories detailed in section 3.2.1 through section 3.2.12. In particular, the structures found by Drescher which utilize

- **Touching what is seen:** The mechanism creates a series of schemas such as *fovf02/SeeHand@22/tactf* which are reliable and give the mechanism the ability to touch an object which is seen through a series of incremental hand actions.
- **Seeing what is touched:** Similarly, *tactf/SeeHand@32/vf33* forms, which gives the mechanism the ability to shift the gaze so as to bring the hand and the touched object into view.
- **Moving the hand into view:** *vp23/hp23/SeeHand@22* is one of a number of schemas which relate various gaze orientations with the ability to see the details of the hand by shifting it to a given orientation.
- **Shifting the gaze to see the hand:** A network of schemas such as *hp33/vp33/SeeHand@22* relate various hand positions with the ability to see the hand by shifting the gaze to a specific orientation.

Table 3.1: Further results of Drescher's CM2 implementation, part one.

goal-directed actions and are detailed in tables 3.1 and 3.2 cannot be duplicated by the reïmplementation, as goal-directed actions are not fully supported by the reimplementation.

3.4 Further results of the reimplementation

The lack of executable goal-directed actions limits the ability of the reimplementation to verify all of the results shown in Drescher's work. However, this does not keep the system from generating many interesting schemas which were not discussed in either [Dre91] or [Dre89].

The analysis program was initially designed to find and categorize only those schemas discussed in section 3.2.1 through section 3.2.12. It also displayed all schemas which were not categorized and had been quite reliable during their life span. By manually examining a number of these lists of schemas, additional categories were added to the analysis program. This process was iterated a few times until categories existed in the analysis program for most of the schemas that were understandable, deemed interesting and occurred in multiple test runs. This process was by no means

- **Persistent positional palpability**: (Try saying that three times fast!) The synthetic item [/hp23/tactl], when on, indicates that there is a palpable object at body relative position 1,3. A series of these synthetic items serve as a map for touchable objects.

- **Persistent positional visibility**: The synthetic item [/vp21/vf14] indicates, when on, that there is a visible object at microworld position 3,5. Again, a series of these synthetic items serve as a map for visible objects.

- **Persistent identity details**: The system also generates synthetic items which can identify certain objects, such as [/hp23/text0] and [/hp02/text3] which denote two different unique objects if *text0* and *text3* do not co-occur between the two objects. Similarly, [/vp23/fovr10] and [/vp23/fovr20] may form and allow the system to tell the difference between two different objects which can both appear at the same position at different points in time (such as the hand and either of the microworld objects).

- **Inversely indexed persistence**: The schema mechanism, for many synthetic items like [/hp23/text0], also builds [/text0/hp23]. To the extent to which *text0* is unique, this serves as an inverse index, where the first structure could be thought as a map indicating what was adjacent to *hp23*, the second structure is a map which says where a specific object is located.

- **Coordinating visible and palpable-object representations**: The creation of various synthetic items culminates in the creation of schemas such as *[/hp23/tactl]/vp23/fovl33* and *[/vp23/fovl03]/hp23/tactl*. The first schema relates that if the world is in the state where an palpable object is at body relative position 1,3, moving the gaze orientation to *vp23* will bring the palpable object into view. The second schema is analogous, but in the other direction, stating that a visible object can be touched.

- **Relational items**: Synthetic items such as *[/vf02/vf10]* relate the position of one object to another, as opposed to the other synthetic items discussed earlier which relate objects to the frame of reference of the body.

Table 3.2: Further results of Drescher's CM2 implementation, part two.

test run	1	2	3	4	5	6	7
total clock ticks	9850	9350	9050	9350	7000	9250	8500
schema saturated	yes		yes	yes			yes
conj saturated		yes			yes	yes	
0: initial	10	10	10	10	10	10	10
reliable	10	10	10	10	10	10	10
1: grasping	8	7	7	7	2	7	8
reliable	6	6	6	6	2	6	6
2: vf/eye/vf shift	51	40	40	43	30	42	43
reliable	49	36	36	36	30	40	40
3: vf shift limit	16	10	3	7	0	6	5
reliable	6	7	2	5	0	5	4
4: fov/eye/fov shift	282	112	153	161	174	156	318
reliable	164	68	115	78	129	117	220
5: vf/eye/fov	198	131	128	124	98	121	162
reliable	171	114	106	104	75	95	125
6: visual network	63	56	59	46	40	47	43
reliable	49	52	51	40	36	43	34
7: hand network	38	41	36	41	25	36	42
reliable	34	36	34	34	24	31	35
8: x ⇒ -x	0	0	0	0	0	0	0
reliable	0	0	0	0	0	0	0
9: hand to body	15	16	15	15	12	20	15
reliable	14	15	15	15	12	20	11
10: seeing hand move vf	1	2	1	3	1	2	0
reliable	1	0	1	2	1	1	0
11: seeing hand move fov	103	26	129	36	28	73	39
reliable	79	21	106	20	28	42	30

Table 3.3: Statistics for Drescher's classifications, test runs 1–7.

test run	8	9	10	11	12	13	14
total clock ticks	8950	8350	9500	8250	8900	8550	10000
schema saturated			yes		yes	yes	
conj saturated	yes	yes		yes			
0: initial	10	10	10	10	10	10	10
reliable	10	10	10	10	10	10	10
1: grasping	8	7	7	4	7	7	8
reliable	7	6	5	3	6	6	7
2: vf/eye/vf shift	46	44	44	36	45	48	42
reliable	39	39	41	32	42	43	40
3: vf shift limit	9	6	8	0	7	4	3
reliable	8	5	6	0	6	3	3
4: fov/eye/fov shift	222	310	257	188	242	311	116
reliable	157	216	181	138	161	222	78
5: vf/eye/fov	135	155	208	127	183	182	132
reliable	118	124	173	112	143	160	119
6: visual network	56	42	54	47	62	49	59
reliable	51	32	44	41	51	39	51
7: hand network	36	32	38	33	34	36	40
reliable	28	28	30	29	29	33	38
8: x ⇒ -x	0	0	0	0	0	0	0
reliable	0	0	0	0	0	0	0
9: hand to body	8	15	20	15	15	16	14
reliable	7	15	18	15	12	14	12
10: seeing hand move vf	0	1	0	3	1	0	1
reliable	0	1	0	3	1	0	0
11: seeing hand move fov	13	32	18	124	44	27	102
reliable	7	26	11	95	38	20	81

Table 3.4: Statistics for Drescher's classifications, test runs 8–14.

test run	15	16	17	18	19	20
total clock ticks	9550	8200	10000	8800	8200	8700
schema saturated	yes				yes	yes
conj saturated		yes		yes		
0: initial	10	10	10	10	10	10
reliable	10	10	10	10	10	10
1: grasping	6	4	7	6	8	8
reliable	5	3	5	5	7	8
2: vf/eye/vf shift	35	40	40	42	36	38
reliable	34	37	38	38	34	38
3: vf shift limit	6	1	6	5	0	7
reliable	5	1	4	4	0	6
4: fov/eye/fov shift	147	79	163	302	223	267
reliable	84	47	118	201	169	211
5: vf/eye/fov	120	121	115	162	140	127
reliable	103	95	104	127	120	116
6: visual network	58	45	59	57	51	54
reliable	54	40	51	48	49	43
7: hand network	42	41	46	37	36	40
reliable	37	34	39	33	31	35
8: x ⇒ -x	0	0	0	0	0	0
reliable	0	0	0	0	0	0
9: hand to body	15	14	15	15	13	11
reliable	15	11	15	14	12	11
10: seeing hand move vf	0	0	2	4	1	1
reliable	0	0	1	1	1	1
11: seeing hand move fov	24	71	32	86	106	3
reliable	15	52	27	57	69	3

Table 3.5: Statistics for Drescher's classifications, test runs 15–20.

category	min	max	median	mean
0: initial	10	10	10	10.0
reliable	10	10	10	10.0
1: grasping	2	8	7	6.65
reliable	2	8	6	5.55
2: vf/eye/vf shift	30	51	42	41.25
reliable	30	49	38	38.1
3: vf shift limit	0	16	6	5.45
reliable	0	8	4	4.0
4: fov/eye/fov shift	79	318	188	209.15
reliable	47	222	138	143.7
5: vf/eye/fov	98	208	131	143.45
reliable	75	173	116	120.2
6: visual network	40	63	54	52.35
reliable	32	54	44	44.95
7: hand network	25	46	37	37.5
reliable	24	39	33	32.6
8: x ⇒ -x	0	0	0	0.0
reliable	0	0	0	0.0
9: hand to body	8	20	15	14.7
reliable	7	20	14	13.65
10: seeing hand move vf	0	4	1	1.2
reliable	0	3	1	0.75
11: seeing hand move fov	3	129	36	55.8
reliable	3	106	28	41.35

Table 3.6: Statistic summary for Drescher's classifications.

exhaustive, and it is certain that there are significant schemas built by the system that are not mentioned in this paper.

3.4.1 Shifting the gaze to see the body

These schemas are similar to the ones which form part of the visual network, but instead of indicating what the resulting gaze orientation is likely to be if an object (most likely the body) is present at a given location and the gaze moved, they indicate, from a series of gaze orientations, which gaze movements will bring the body into view, and where it will be seen. A typical schema built by the reimplementation is *vp11/eyel/vf30* (illustrated by figure 3-1). The format recognized by the analysis program is *vp??/eye?/vf??* where the elements of the schema must relate correctly to one another and the object seen by the coarse visual item must be the body for it to be included in this category.

Another related, but separate category, contains schemas which bringing the body into detailed view from a given gaze orientation. A typical schema is *vp11/eyeb/fovb20* (illustrated in figure 3-2). The analysis program uses a similar format and has similar requirements for this category as it does for the prior category.

3.4.2 Using the body as a visual position reference

These schemas do the opposite indexing as the previous category, as when the body is seen in a foveal region and recognized, moving the gaze in a given direction will result in a specific final gaze orientation. A typical schema is *fovb20/eyer/vp20.*

This category also includes schemas with the foveal item negated, as long as the visual position item is also negated, as in the schema *-fovb20/eyef/-vp11.* This schema is actually a very strong statement, as it indicates if *fovb20* is off, moving the gaze forward will *never* result in a gaze orientation of *vp11.* In other words, this schema says that *fovb20* is *always* on whenever the gaze is oriented at *vp10*, as opposed to the less constrained schema *fovb20/eyer/vp20* which merely says that if *fovb20* is on, and the gaze is moved right, the resulting gaze orientation makes *vp20* on. This second schema, even if highly reliable, admits the possibility of the existence of other schemas with the same action and result and completely different context items. The first schema, if highly reliable, makes a very strong statement about the conditions required to reach the result state.

The analysis program recognizes schemas of either format *fov???/eye?/vp??* or *-fov???/eye?/-vp??* as belonging to this category provided that the items relate correctly to one another and the object seen is, in fact, the body.

3.4.3 Hand network constraint schemas

Distinctly related to the hand network schemas, these schemas indicate which transitions are not possible. A typical schema is *-hp21/handl/-hp11*. Again, these schemas, when reliable, make stronger statements about the microworld than the previously discussed hand network schemas. In particular, this schema says that *hp11* is only reachable through a *handl* action if the hand is at *hp21* before the action is taken. Schemas which belong to this category must satisfy precisely the same requirements as for the hand network schemas, except that both items are negated.

3.4.4 Visual network constraint schemas

A similar set of constraint schemas exist for the visual network. *-vp02/eyeb/-vp01* is a schema which is built by the reimplementation and included by the analysis program in this category. The analysis program looks for schemas which match the format *-vp??/eye?/-vp??* and have the correct relationship between the various components.

3.4.5 Coarse visual shift constraint schemas

-vf??/eye?/-vf?? schemas, if the coarse visual items relate correctly between one another and the gaze action, are placed in this category by the analysis program. These schemas give an alternate perception of the coarse visual field shift network described earlier. *-vf11/eyef/-vf10* is a typical schema built by the reimplementation

which indicates that *vf10* cannot be turned on by the *eyef* action if *vf11* is off before the action is taken.

3.4.6 Detailed visual shift constraint schemas

These schemas give a different description of the foveal region shift network described earlier. If the foveal regions relate to one another through the gaze action correctly, the analysis program puts all schemas of the format *-fov???/eye?/-fov???* in this category. *-fovx01/eyeb/-fovf01* is a typical schema, however, the two details need not be identical. (Note: this category appears as category 24 in tables 3.7, 3.8 and 3.9, not in the same order as these categories are presented here.)

3.4.7 Coarse to detailed visual shift constraint schemas

-vf31/eyer/-fovb10 is a typical member of this category. The analysis program looks for schemas which match this format and have a gaze action which moves the object at a given coarse visual coordinate to a foveal region.

3.4.8 Detailed to coarse visual shift constraint schemas

-fovr02/eyef/-vf31 is a typical member of this category. The analysis program looks for schemas which match this format and have a gaze action which moves the object in a given foveal region to a specified coarse visual coordinate.

3.4.9 Detailed to coarse visual shift schemas

Strangely enough, neither [Dre91] nor [Dre89] report any schemas of the format *fov???/eye?/vf??*, even though the development of these schemas occurs in parallel with the coarse to detailed visual shift schemas which were reported in these sources. Figure 3-1 illustrates a situation where the schema *fovb02/eyel/vf31* has been activated and successful due to the shifting appearance of the hand. The analysis program has the same requirements for membership in this category as for the preceding one, except that the items are not negated.

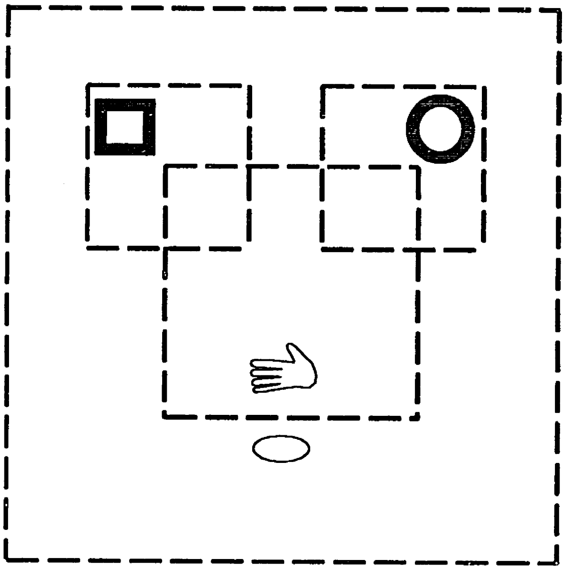

Figure 3-5: Through a large series of schemas, the reimplementation code learns that there are three different objects which each can appear in specific regions of the microworld.

3.4.10 Seeing objects in different visual regions

The reimplementation builds a number of very reliable schemas which match one of the formats *vp??/eye?/vf??*, *vp??/eye?/fov???*, *-vp??/eye?/-vf??* or *-vp??/eye?/ -fov???*. Examined separately, each schema makes some statement describing where objects appear in the microworld. Taken together, these schemas seem to describe which areas of the microworld each object may appear in. In particular, the left hand and right hand objects are each constrained to be at one of four microworld positions, and the hand is constrained to be in one of nine microworld positions. (The body is, of course, immobile and is only seen in one position.) These three regions where the three movable objects can be seen are effectively separated and defined by the series of schemas defined by the mechanism. (See figure 3-5.) The analysis program accepts schemas which match any of the four formats provided that the components relate to one another correctly.

3.4.11 Hand to body constraint schemas

The hand position in front of the body is only reachable from certain hand positions. *-hp20/handl/-taste1* is a typical schema which belongs to this category. The analysis program has the same requirements for these schemas as it does for the hand to body schemas, except that the context and result items must be negated.

3.4.12 Hand movement against object

The reimplementation learns that an object which is felt by the hand cannot be pushed out of the way and indicates this knowledge by building the schemas *tactr/handr/tactr*, *tactl/handl/tactl*, *tactb/handb/tactb* and *tactf/handf/tactf.* The system also builds schemas which have various *text?* items substituted in positions where the *tactl* makes sense, as the former items are detailed versions of the latter coarse item.

These schemas work even when the hand is grasping an object. If the hand is moving left, the object moves with the hand, and the *tactl* or *text?* item is still on after the action is taken. If the hand is moving in any other direction, the object in that direction still blocks the movement of the hand.

The analysis program recognizes each of the four schemas noted above, plus variations where a *text?* item is substituted for a *tactl* item.

3.4.13 Backward hand movement against body

Similarly, the reimplementation learns about situations where the hand is in front of the body and moved backward against it. A series of schemas of the format *(taste? or tactb or bodyf)/handb/(taste? or tactb or bodyf)* are created which represent the various items which are on whenever the hand is in front of the body. Schemas which match this format are placed in this category by the analysis program.

3.4.14 Hand movement from coarse visual to foveal region schemas

[Dre91] and [Dre89] mention discovery of *vf??/hand?/vf??* and *fov???/hand?/fov??* schemas by the CM2 implementation, but they do not mention the construction of *vf??/hand?/fov???* schemas. This category is included for completeness, and contains those schemas which match this format and have the correct relationships between components.

3.4.15 Support data from test runs

The twenty test runs were also analyzed for the categories detailed in this section. The data gathered is presented in tables 3.7, 3.8 and 3.9. It is summarized in table 3.10. Discussion of this data and conclusions can be found in chapter 4.

test run	1	2	3	4	5	6	7
total clock ticks	9850	9350	9050	9350	7000	9250	8500
12: vp/eye/vf body	17	14	16	18	12	11	12
reliable	16	12	16	15	11	10	11
13: vp/eye/fov body	31	37	27	47	25	24	25
reliable	29	35	20	47	25	20	25
14: fov body/eye/vp	8	6	12	10	5	11	9
reliable	7	4	7	6	5	9	8
15: hp required for hp	10	5	12	5	11	12	6
reliable	10	5	11	5	10	11	6
16: vp required for vp	5	6	6	7	6	7	9
reliable	5	6	6	7	6	7	9
17: vf required for vf	22	26	29	27	25	30	29
reliable	20	22	27	23	21	26	26
18: vf required for fov	71	78	101	84	102	115	91
reliable	61	70	97	68	86	108	77
19: fov required for vf	18	28	35	23	26	12	26
reliable	12	21	28	11	20	9	22
20: fov/eye/vf	77	50	46	53	40	58	61
reliable	54	38	34	33	37	41	43
21: visual regions for objects	53	71	50	45	102	57	51
reliable	43	62	43	31	90	54	41
22: hp required for body	5	4	5	5	5	0	5
reliable	5	4	5	5	5	0	5
23: hand against object	8	7	7	6	1	6	8
reliable	6	6	7	5	1	5	4
24: fov required for fov	64	142	197	136	182	68	75
reliable	50	109	142	95	129	56	60
25: hand against body	20	24	22	15	17	12	22
reliable	12	19	19	10	13	9	12
26: hand movement vf ⇔ fov	44	25	37	61	8	53	15
reliable	30	20	34	40	3	33	10

Table 3.7: Statistics for new classifications, test runs 1–7.

test run	8	9	10	11	12	13	14
total clock ticks	8950	8350	9500	8250	8900	8550	10000
12: vp/eye/vf body	15	13	15	13	20	11	16
reliable	11	13	11	11	15	9	15
13: vp/eye/fov body	36	25	25	25	47	24	36
reliable	32	21	25	24	45	23	29
14: fov body/eye/vp	6	9	8	9	9	7	8
reliable	4	6	4	5	8	6	6
15: hp required for hp	8	11	8	11	11	7	6
reliable	8	9	8	11	10	7	6
16: vp required for vp	7	6	8	5	5	6	5
reliable	7	6	8	5	5	6	5
17: vf required for vf	23	26	26	24	27	21	29
reliable	22	25	23	24	23	18	23
18: vf required for fov	85	96	65	75	79	71	82
reliable	70	89	56	73	73	54	64
19: fov required for vf	32	16	43	10	23	19	45
reliable	21	12	32	8	17	16	31
20: fov/eye/vf	54	61	43	45	63	63	35
reliable	43	46	30	36	48	45	26
21: visual regions for objects	51	102	66	95	67	70	74
reliable	44	84	52	84	40	49	50
22: hp required for body	9	5	0	5	5	4	5
reliable	9	5	0	5	5	4	5
23: hand against object	8	3	3	2	8	7	7
reliable	5	1	1	2	8	6	7
24: fov required for fov	148	122	135	81	147	104	209
reliable	93	86	78	71	109	70	147
25: hand against body	8	17	15	20	19	10	24
reliable	6	12	8	17	13	3	21
26: hand movement vf ⇔ fov	39	9	29	41	24	16	49
reliable	17	3	22	32	15	13	27

Table 3.8: Statistics for new classifications, test runs 8–14.

test run	15	16	17	18	19	20
total clock ticks	9550	8200	10000	8800	8200	8700
12: vp/eye/vf body	16	9	17	15	14	13
reliable	14	7	15	15	13	11
13: vp/eye/fov body	35	22	31	45	37	4
reliable	34	22	26	45	37	4
14: fov body/eye/vp	14	11	8	8	8	8
reliable	11	10	2	7	7	7
15: hp required for hp	6	6	5	11	6	7
reliable	6	6	5	8	6	7
16: vp required for vp	7	9	6	7	6	7
reliable	7	9	6	7	6	7
17: vf required for vf	34	30	30	30	36	34
reliable	31	28	30	25	34	31
18: vf required for fov	108	124	122	85	98	110
reliable	100	109	99	58	77	100
19: fov required for vf	26	29	38	32	53	40
reliable	20	25	27	22	37	31
20: fov/eye/vf	54	37	38	47	29	49
reliable	41	23	31	33	20	42
21: visual regions for objects	47	62	62	29	41	70
reliable	43	48	46	19	32	59
22: hp required for body	5	5	5	5	5	9
reliable	5	5	5	5	5	9
23: hand against object	7	6	6	3	7	8
reliable	7	6	6	2	7	7
24: fov required for fov	111	223	134	36	207	161
reliable	85	165	95	21	166	131
25: hand against body	24	10	23	18	12	18
reliable	22	9	18	12	11	12
26: hand movement vf ⇔ fov	17	46	39	63	31	29
reliable	11	42	21	51	25	15

Table 3.9: Statistics for new classifications, test runs 15–20.

category	min	max	median	mean
12: vp/eye/vf body	9	20	14	14.35
reliable	7	16	12	12.55
13: vp/eye/fov body	4	47	27	30.4
reliable	4	47	25	28.4
14: fov body/eye/vp	5	14	8	8.7
reliable	2	11	6	6.45
15: hp required for hp	5	12	7	8.2
reliable	5	11	7	7.75
16: vp required for vp	5	9	6	6.5
reliable	5	9	6	6.5
17: vf required for vf	21	36	27	27.9
reliable	18	34	24	25.1
18: vf required for fov	65	124	85	92.1
reliable	54	109	73	79.45
19: fov required for vf	10	53	26	28.7
reliable	8	37	21	21.1
20: fov/eye/vf	29	77	49	50.15
reliable	20	54	37	37.2
21: visual regions for objects	29	102	62	63.25
reliable	19	90	46	50.7
22: hp required for body	0	9	5	4.8
reliable	0	9	5	4.8
23: hand against object	1	8	7	5.9
reliable	1	8	6	4.95
24: fov required for fov	36	223	135	134.1
reliable	21	166	93	97.9
25: hand against body	8	24	18	17.5
reliable	3	22	12	12.9
26: hand movement vf ⇔ fov	8	63	31	33.75
reliable	3	51	21	23.2

Table 3.10: Statistic summary for new classifications.

Chapter 4

Analysis, Discussion and Conclusions

4.1 Test run evidence confirms Drescher's results

Drescher, in [Dre91], discusses results which are found by the CM2 implementation which this paper splits into 22 different categories. Of these categories, 10 are unable to be built by the reimplementation code, due to the fact that the reimplementation does not fully implement goal-directed actions. The other 12 categories formed the basis of the analysis program.

Table 3.6 summarizes the results of analyzing all twenty test runs for these 12 schema categories. As a whole, the test runs support the results found by Drescher in 11 of the 12 categories.

All twenty test runs built */grasp/hcl* and further, each built */ungrasp/-hcl.* 19 build */grasp/hgr*, and 18 of these build *tactl/grasp/hgr.* Each of these schemas is mentioned by Drescher, and this is excellent verification of these results.

Beyond these results, 16 test runs have one or more schemas of the form *text?/grasp/hgr* which have the ability to discern between various objects, as the body is immobile and cannot be grasped successfully, and *tactl/grasp/hgr* is therefore not very reliable. 17 test runs build */ungrasp/-hgr*, which is closely related to */grasp/hgr* and */ungrasp/-hcl.*

There was another interesting result, too, one which is not very intuitive. *-hcl/grasp/hcl* is built in 9 test runs. This schema indicates the prerequisite that the hand cannot be already closed when the grasp action is taken for it to become closed reliably, as very occasionally, if the hand is closed and the grasp action taken, the call to clock tick forces the hand open as the number of consecutive clock ticks with the hand closed is three. This indicates how good the marginal attribution algorithm is, actually, as this schema indicates understanding the necessity of opening the hand first if one wants to close the hand and keep it closed for a while reliably.

All twenty test runs build at least 30 reliable coarse visual item shifting schemas. The coarse visual item shift limit schemas, however, are quite a bit more rare. In three of the twenty test runs, not a single shift limit schema was formed. This is probably due to the fact that, in order to form, the system must both repeatedly try to shift the gaze from a particular extreme position in a particular direction, and the object which is seen via the coarse visual item must be in the same position. In detail, to have *vf11/eyer/vf01* spinoff *-vp20&vf11/eyer/vf01*, the system must believe that creating a child schema with *-vp20* added will make a more reliable version of this schema. The marginal attribution algorithm, to come to this conclusion, must see a series of trials which alternate between having *vp20* on and off with the parent schema activated (i.e. *vf11* on and *eyer* action taken). The series of trials must be at least fifteen in length with at least eight *vp20* off trials where the result obtains and at least seven *vp20* on trials where the result does not obtain in order to have the system decide it is a significant context item in the negative direction. These are a pretty serious group of requirements, which is most likely why not very many of these schemas are built in each test run.

While the CM2 reference run builds a total of 79 coarse visual field schemas (including those that reference a visual position to the position of the body), the test runs, on average, only built 41 such schemas. This discrepancy may be due to a misunderstanding. Neither [Dre91] or [Dre89] specify if schemas like *vf00/eyel/vf00* should be included in this category. The analysis program requires that the two items be different and related to one another correctly via the gaze action in order to be placed in this category, which eliminates schemas such as the one above. Another possibility is that the alternation between random and goal-directed behavior in the toplevel control loop for the CM2 implementation might serve to bias the system towards discovering more about coarse visual items, as they are some of the first goal-directed actions built, there would be a natural bias towards these types of actions.

Massive numbers of foveal shift schemas are built in the reimplementation test runs. A number of these schemas are unreliable, as the system does not know which details co-occur between objects, and has to discover this. However, most of the test runs also have a number of schemas such as *fovf01&fovf02/eyef/fovx32* where the system is clearly trying to improve the reliability of the result by defining precisely which object is meant by the context. As the body can only appear in the rear foveal region (*fovb* items), it cannot be referred to by this schema, and in fact, as it cannot be shifted into any other foveal region, it can never be the subject of a schema of this type. Interestingly enough, of the three other objects in the world, only two objects satisfy the context when in the front foveal region, and these two objects both have detail 32 on, so this schema is reliable provided that the gaze isn't oriented at an extreme position. This is an improvement on its parent schema, *fovf01/eyef/fovx32*, whose context is satisfied by every object in the microworld but whose result only follows reliably for two objects. This process of iterative additions to the context eventually culminates in schemas with contexts which match only those objects which have the particular visual detail referenced in their results. *fovr00&fovr02/eyer/fovx03* is a good example, as its context can only be satisfied by the hand (as the body cannot appear in the right foveal region) and the hand has visual detail 03. Context items may also be negated, for example, *fovl01&-fovl20/eyel/fovx11* has a satisfied context when either the hand or the right hand object is in the left foveal region. Again, both objects have visual detail 11 on, and so further identification of each object is unnecessary. As noted earlier, however, a good number of intermediate schemas which are unreliable due to incorrect detail relationships must be built in order to build reliable schemas with more specific contexts. All visual shift schemas may also be unreliable due to the fact that they do not take into account the limits of the gaze orientation, as mentioned earlier.

Every test run also has a number of schemas which concern shifting an object from a coarse visual item position into a foveal region. While most of these schemas in each test run are of the format *vf??/eye?/fov???*, at least one in each test run includes a detailed foveal item added to the context, such as *vf12&fovl00/eyel/fovx12*. The context in this schema, to improve reliability, discerns between the two objects which can appear at *vf12*, the hand and the left object, where the hand has both details 00 and 12 and the left hand object has neither. Through this series of schemas the system is again relating precisely which objects have the various visual detail items on. However, most of these schemas deal with the more common situation where the details of the object are not apparent until the object is shifted into a foveal region. In these situations, the schemas which are most reliable are those which have a visual detail result which is shared among all of the objects which can appear in that foveal region.

The numbers reported for the visual network schemas in table 3.6 don't quite match with those reported in [Dre91], where 17 visual network schemas and 7 pairs of schemas which relate a coarse visual item seeing the body (and visual position, for the other schema in the pair) to a given visual position were described with non-identical visual position items. As mentioned earlier, however, the analysis program accepts schemas where the two visual position items are identical as long as they are at an extreme where the gaze action will not change the gaze orientation, which definitely increases the number of schemas belonging to this category.

To get a better idea of what the numbers in the table for this category represent, the first test run was examined directly. The analysis program found 63 visual network schemas in the first test run. 19 schemas were traditional visual network schemas as reported in [Dre91] such as *vp12/eyer/vp22*, where the two visual position items are different from one another. There were a number of schema pairs as discussed in [Dre91], however, many of these pairs covered material which was already adequately covered by other schemas. Only one schema pair was found which effectively added a segment of the network which was missing. A total of another 5 schema pairs were found, but they did not add anything new to the network. 12 schemas were the complete set of limit schemas such as *vp00/eyel/vp00*. 7 schemas dealt with seeing the body when at an extreme gaze orientation, such as *vf31/eyel/vp00* where the glance orientation doesn't change. Another 5 schemas combined these two ideas together, adding coarse visual information to limit schemas, such as *vp21&vf10/eyer/vp21*, where the body appears at *vf10* when the gaze is oriented at *vp21*. This brief analysis indicates that the system did, in fact, succeed in acquiring a substantial body of knowledge about the visual position network. A quick analysis of a few of the other test runs again yielded data which supported an estimate of roughly 80% of the 24 possible visual position schemas are built in each test run. Similarly, roughly 80% of the hand position network is constructed in each of the test runs.

The system was very good at building schemas which relate information about how to move the hand to the body. In most cases, the system built over 75% of the 20 possible schemas of the form *hp??/hand?/(taste? or tactb or bodyf)*. (20 schemas are possible as the hand has three taste details and can be moved in four different directions.) In two of the twenty test runs, the entire group of schemas is built.

The test runs rarely built schemas which related the situation where the movement of the hand was seen via the coarse visual items. [Dre91] makes it clear that these schemas were uncommon in its reference run. This, coupled with the fact that the system was able to see the hand move in other modes, leads to the conclusion that the lack of *vf??/hand?/vf??* schemas in the test runs is not notably unusual. In particular, the schema mechanism was usually successful in finding a number of schemas which related the hand shifting between various foveal regions. The formation of these *fov???/hand?/fov???* schemas seemed to be very dependent on the particular test run, however. These schemas will not form if the hand is rarely moved around in the foveal regions. A more likely reason for a lack of these schemas is that the hand, when in a foveal region, was often in an extreme position and the hand action didn't result in a new hand position. When the system was successful in finding these schemas in a given test run, it even built schemas such as *fovx01&fovx03/handl/fovl32*

where the context accurately picks the hand out of all objects which can appear in the center foveal region.

One category was basically not confirmed by the reimplementation, that of negated consequence schemas. In this particular case, [Dre91] and [Dre89] disagreed in what was expected to form a negated consequence schema, and the more constrained format of *x/(eye? or hand?)/-x* shown in [Dre89] was used. Not a single schema of this format was found in any of the test runs. However, examination by hand showed that many schemas of the format given in [Dre91] were formed, in fact, for many of the schemas in the other categories, */(eye? or hand?)/-x* schemas are required precursors to the formation of the appropriate child schemas.

4.2 Analysis of all results

The division between Drescher's results and those which were discussed in section 3.4 is an artificial one. With the massive amounts of data, and Drescher's interest in spending most of his time explaining and discovering the most complex structures the CM2 implementation built in its reference run, it is likely that the CM2 implementation did build some of each of the schemas first discussed in this work, but that their significance wasn't commented upon. However, these schemas give a much stronger body of proof of the ability of the marginal attribution algorithm to find and codify regularities in the microworld.

With the more detailed results given above, it's clear that the system has a good grasp of how the hand actions work. (Apologies for the awful pun...I just couldn't restrain myself.) In particular, the *-hcl/grasp/hcl* schema, built in nine of the twenty test runs, shows an unexpectedly pure insight into the mechanics of the microworld. The microworld, if the hand is closed, keeps a count and only allows the hand to remain closed for three time units. The extended context of */grasp/hcl* keeps track of each item to see if it helps predict the successful activation of the schema. In particular, it is determined that *hcl* has the peculiar effect of needing to be off in order for the result of having *hcl* on occur more reliably. This is precisely because occasionally, when *hcl* is on and the *grasp* action taken, taking the action doesn't change the situation at all, and the call to clock tick which occurs after each action is taken results in the hand being opened, as it has been closed for three clock ticks. The system learns that the hand ends up closed most reliably when it is open immediately before the grasp action is taken. To amplify this, if the system had a goal of ending up with the hand closed, it would have the knowledge necessary to recommend opening the hand first, and then closing it, which reliably gives the desired result.

By any reasonable measure, the system clearly understands the relationships between its various visual items and the glance actions. In every test run, it generates hundreds of schemas which relate coarse visual items to one another, detailed visual items to one another, coarse visual items to detailed visual items, and vice versa. It also learns how to relate sensed visual positions (through the visual proprioceptive items or from seeing the body at a given coarse visual item position or detailed foveal region) to other visual positions through glance actions. Each of these relationships also explore the abilities and limits of the visual system, through indications of how to view the visual details of an item by shifting the gaze since the foveal area is limited, as well as through schemas which indicate the limits of the gaze orientation.

These relationships are further strengthened by groups of *constraint* schemas. These schemas, first discussed in this work, help to define the various networks discussed in the preceding paragraph by indicating the items required for a given transition to take place. Each test run found a number of constraint schemas corresponding to each of the types mentioned above — *-vf??/eye?/-vf??*, *-fov???/eye?/-fov???*, *-vf??/eye?/-fov???*, *-fov???/eye?/-vf??* and *-vp??/eye?/-vp??*. These schemas indicate, in a very strong way, which transitions involving visual items are *impossible* in the microworld, given certain situations. *-vf11/eyef/-vf10* stands in contrast with its relative *vf11/eyef/vf10*, as the first schema, if reliable, indicates that none of the other coarse visual items have anything to do with the state of *vf10* when the *eyef* action is taken; if *vf11* is off, then *vf10* always ends up off. Essentially, when this schema is reliable, it makes a statement about all 25 of the visual items at once, as 24 of the items are apparently unrelated to achieving the result, and only one item is related to the result. The second schema, when reliable, still admits the possibility of other schemas with different contexts which do not include *vf10* and are also reliable. These constraint schemas are a major development by the schema mechanism; when taken along with all other related schemas, they indicate an understanding which seems to go beyond any sort of simple cause and effect nature.

Similarly, but more simply, the system has a substantial body of knowledge about the relationship between hand position and the hand actions, as well as between hand positions and the position of the body. The system learns both *hp??/hand?/hp??* and constraining *-hp??/hand?/-hp??* schemas, effectively representing a network which relates the various hand positions to one another through the hand actions. Gaps in this network are filled in by relationships between hand positions and tactile feedback from having the hand in front of the body. The system, by exploiting these relationships, is able to move the hand to the body if it is in an adjacent position. It is also able to predict which position the hand will end up in after it is moved when the system can feel the hand in front of the body through the tactile items. It is also able to understand the constraints involved, as the tactile feedback unique to having the hand in front of the body can only be achieved by moving the hand into the correct position; no other situation can satisfy these constraints.

The system also gains the ability to understand that pushing its hand against an object which it can feel is futile; the object does not move. In the case of pushing against the body, the system also learns that the hand does not change position and remains in front of the body, as well as learning precisely which items are on when the hand is in front of the body.

The abilities of the system aren't just relating the senses and actions having to do with a particular faculty to one another. It is able to, and in fact does, relate the visual and hand systems to one another. *vf??/hand?/vf??*, *fov???/hand?/fov??? vf??/hand?/fov???* and *fov???/hand?/vf??* schemas all form to indicate the visible result of a hand motion

The system also learns a lot about objects in the microworld. It can predict where the image of the body will appear in response to a given gaze movement from schemas such as *vp??/eye?/vf??* and *vp??/eye?/fov???*. It also can use the visual perception of the body as a way of determining the current gaze orientation. *fov???/eye?/vp??* schemas can give this reliably when the foveal detail item is one which is included only by the body. In addition, the system effectively creates a map of the microworld with an understanding of the regions in which each object can appear. This map is created over time, and is represented as a series of *vp??/eye?/vf??*, *vp??/eye?/fov???* and their corresponding constraint schemas which collectively indicate which objects (and with which detail items) appear at each region of the microworld. The body location schemas mentioned earlier are similar and can be thought of as a special case where the object is restricted to a one unit area and therefore does not move.

In summary, the system displays an incredible amount of understanding about the world, creating a large body of knowledge merely by observing what happens when actions are taken. This is especially notable when it is remembered that the system starts out with virtually *no* information about the microworld whatsoever, just a set of blank schemas, one for each action. It does not know which items relate to one another or to a particular action; the names by which each of these is referred to in this paper is merely a series of names which are given to the items and actions by convention and are only human readable. The system itself discerns between items strictly by item number and between actions by action number. Given this impoverished start, its accomplishments are even more notable.

4.3 Performance

Direct examination of the timing data generated when executing the test runs indicates that the length of time required to process each clock tick increases linearly with the number of schemas in the system. This is most likely largely due to the serial implementation of the system, as each schema must be updated in sequence throughout the process. A parallel implementation may be able to achieve substantially better performance if carefully coded. The current implementation finishes a test run to 10000 clock ticks in roughly 2 1/2 days of CPU time on a DECStation

5000/120 with 16 megabytes of memory. However, as has been noted earlier, faster UNIX workstations which run Lucid LISP already exist and will continue to be developed. Indeed, if the current code was ran on the fastest available UNIX machine, it would probably finish in roughly 12 hours, as an estimate. This is one big advantage enjoyed by this implementation — as better hardware and Common LISP software becomes available, this system can take advantage of it without requiring anything but minor changes.

In addition, it is almost certain that a clever algorithm designer could find ways of improving the speed of certain crucial portions of code by combining various steps, building new data structures to speed up certain portions of the algorithm, or perhaps even coming up with a completely different but functionally equivalent algorithm.

4.4 Future directions

This work, while begun as an attempt to verify Drescher's results, is now incredibly open-ended. The possibilities for future work based upon this one are endless, perhaps even more so as the implementation uses Common LISP and future researchers are strongly encouraged to use and modify the code which is included in appendix A for this purpose.

These suggestions are by no means exhaustive, and range approximately from the simple to the complex.

- Run the code on a faster UNIX workstation with more memory, and see if the system learns more in a longer test run.

- Find more schema categories and add them to the analysis program, see which of them show up in most or all of the test runs. See if there is a strong correspondence between what schemas are built in a given category and which are built in another category; it is entirely possible that there are inter-category dependencies, and this hasn't been examined at all.

- Try running the code with fewer primitive items in the microworld, and see how this affects the types and numbers of the schemas built. Do the same with a richer microworld with more primitive items. Try decreasing or increasing the number of possible gaze orientations or hand positions.

- Improve the analysis program so the system can compare test runs to see exactly which schemas were built in every test run. Use this data to further refine the categories and the overall impression of what knowledge the system learns about the microworld. Also use this data to develop a full picture of when the system develops each category.

- Analyze the microworld rigorously, deciding which regularities should be able to be learned and how complex learning each of the regularities should be (i.e. how often is there a counterexample, and how often does the given situation occur?) Give a theory for which categories should exist, and in which order they should be created. See if the mechanism can build all of these categories, and if other categories are discovered, use this to redo the analysis of the microworld.

- Figure out a good way of storing the controller data required for execution of the goal-directed actions. Complete the implementation of goal-directed actions, remove the accessibility tests and instead use the method given in [Dre91] to decide when a new goal-directed action should be built.

- With goal-directed actions complete, make them executable, adding appropriate controls to make sure the system does not favor one set of actions over another. See if the rest of Drescher's results can be verified. If not, why not?

- Analyze the algorithms used in the system and redesign them for greater efficiency.

Appendix A

Source code

A.1 Microworld simulator

```
;;; MICRO.LISP
;;; Microworld implementation
;;; Part of the reimplementation of the CM2 implementation
;;; as defined and explained in
;;; "Made-up Minds: A Constructivist Approach to Artificial Intelligence"
;;; by Gary L. Drescher, as published by MIT Press 1991
;;; originally published as a Ph.D. thesis September 1989
;;; Massachusetts Institute of Technology

;;; Part of the Computer Science Masters' thesis for
;;; Robert Ramstad, September 1992
;;; Massachusetts Institute of Technology
;;; Thesis Advisors: Professor Ronald Rivest
;;;   and Bruce Foster, Digital Equipment Corporation
;;; Copyright (c) 1992, Robert Ramstad, All Rights Reserved.
;;; The author hereby grants to MIT permission to reproduce and to
;;; distribute copies of this document in whole or in part.

;;; this file implements a simple software simulation of a world
;;; it contains variables, definitions and functions for:
;;; objects in the world and their characteristics
;;; 140+ functions which can be called to get the status of each
;;;   primitive item
;;; 17 functions which correspond to the primitive actions
;;; clock-tick (a function which randomizes object position,
;;;   opens the hand after 3 time units of being closed,
;;;   and increments the clock)
;;; keeping the state of the world, initializing it, and loading/saving
;;;   the random number generator seed

;;; note: the state of the microworld is kept internally, nothing
;;; other than t/nil/status messages are returned by any function

(in-package 'microworld)
```

```
(export
 '(hp00 hp01 hp02 hp10 hp11 hp12 hp20 hp21 hp22

   vp00 vp01 vp02 vp10 vp11 vp12 vp20 vp21 vp22

   vf00 vf01 vf02 vf03 vf04
   vf10 vf11 vf12 vf13 vf14
   vf20 vf21 vf22 vf23 vf24
   vf30 vf31 vf32 vf33 vf34
   vf40 vf41 vf42 vf43 vf44

   fovf00 fovf01 fovf02 fovf03
   fovf10 fovf11 fovf12 fovf13
   fovf20 fovf21 fovf22 fovf23
   fovf30 fovf31 fovf32 fovf33
   fovb00 fovb01 fovb02 fovb03
   fovb10 fovb11 fovb12 fovb13
   fovb20 fovb21 fovb22 fovb23
   fovb30 fovb31 fovb32 fovb33
   fovl00 fovl01 fovl02 fovl03
   fovl10 fovl11 fovl12 fovl13
   fovl20 fovl21 fovl22 fovl23
   fovl30 fovl31 fovl32 fovl33
   fovr00 fovr01 fovr02 fovr03
   fovr10 fovr11 fovr12 fovr13
   fovr20 fovr21 fovr22 fovr23
   fovr30 fovr31 fovr32 fovr33
   fovx00 fovx01 fovx02 fovx03
   fovx10 fovx11 fovx12 fovx13
   fovx20 fovx21 fovx22 fovx23
   fovx30 fovx31 fovx32 fovx33

   tactf tactb tactr tactl
   bodyf bodyb bodyr bodyl

   text0 text1 text2 text3
   taste0 taste1 taste2 taste3

   hcl hgr
   always-true always-false

   handf handb handr handl
   eyef  eyeb  eyer  eyel
   grasp ungrasp

   clock-tick
   new-initial-random-state
   init-world
   init-world-no-right-object
   init-world-no-left-object

   test1-init-world
   test2-init-world
   test3-init-world))
```

```
(use-package 'fix)

(proclaim '(fixnum *hp-x*   *hp-y*
                   *vp-x*   *vp-y*
                   *clock*  *grip-expiration*))

(proclaim '(atom   *hcl*    *hgr*))

(proclaim '(type (simple-array t (7 7)) *world*))

;;; note on coordinate systems:
;;; there are three coordinate systems which are used within this
;;; file: microspace, body and glance relative coordinates
;;; in each case, the X axis (first position in the coordinate pair)
;;; runs left to right while the Y axis (second position) runs bottom
;;; to top, i.e. traditional mathematical notation

;;; microspace relative coordinates reference the 7x7 world array
;;; directly where the lower left hand corner is position (0,0) and
;;; the lower right hand corner is position (6,0)

;;; body relative coordinates are often used when referring to the
;;; center 3x3 area in the microworld
;;; the lower left corner of this area is microspace coordinate (2,2)
;;; which is defined as body relative coordinate (0,0)

;;; glance relative coordinates are used when referring to the 5x5
;;; area centered around the current glance orientation
;;; the current glance orientation is stored as microspace relative
;;; coordinates in *vp-x* and *vp-y*
;;; the center of the 5x5 glance relative area is defined as glance
;;; relative coordinate (2,2) with the lower left corner of this area
;;; defined as glance relative coordinate (0,0)

;;; the world is modeled as a 7x7 "microspace" which is
;;; represented as a 2D array, with indices 0 through 6
```

world

```
(defvar *world*
  (make-array '(7 7) :initial-element nil))

;;; brief accessor for readability
```

get-object

```
(defmacro get-object (x y)
  '(aref *world* ,x ,y))
```

```
;;; world-object datatype

;;; world objects have three sets of characteristics:
;;; visual, tactile and taste, each of which is represented as
;;; an array of t or nil elements
;;; in addition, objects can be marked either movable or not,
;;; currently, only the body and hand are considered to be immovable
;;; objects
;;; movable objects can be moved by the hand, and if not currently
;;; grasped will be randomly moved occasionally by the clock-tick
;;; function in a direction indicated by the current cycle value
;;; 0 indicates the next move is to the right, 1 down, 2 left, 3 up
;;; i.e. the objects move in a clockwise direction over time
```

world-object

```
(defstruct world-object
  (visual
   (make-array 16 :initial-element nil)
   :type (simple-array atom (16)))
  (tactile
   (make-array 4 :initial-element nil)
   :type (simple-array atom (4)))
  (taste
   (make-array 4 :initial-element nil)
   :type (simple-array atom (4)))
  (movable t :type atom)
  (cycle 0 :type fixnum))

;;; brief accessors for readability
```

get-visual

```
(defmacro get-visual (object index)
  (declare (fixnum index))
  '(the atom (svref (world-object-visual ,object) ,index)))
```

get-tactile

```
(defmacro get-tactile (object index)
  (declare (fixnum index))
  '(the atom (svref (world-object-tactile ,object) ,index)))
```

get-taste

```
(defmacro get-taste (object index)
  (declare (fixnum index))
  '(the atom (svref (world-object-taste ,object) ,index)))
```

get-movable

```
(defmacro get-movable (object)
  '(the atom (world-object-movable ,object)))
```

get-cycle

```
(defmacro get-cycle (object)
  '(the fixnum (world-object-cycle ,object)))
```

```
;;; defining the body and hand objects
```

body

```
(defvar *body*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(t   t   t   t
                           t   nil nil t
                           t   nil nil t
                           t   t   t   t))
   :tactile  (make-array 4
                         :initial-contents
                         '(t   t   nil nil))
   :taste    (make-array 4
                         :initial-contents
                         '(t   nil nil nil))
   :movable nil))
```

hand

```
(defvar *hand*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(t   t   t   t
                           t   t   t   t
                           nil nil t   t
                           t   t   t   t))
   :tactile  (make-array 4
                         :initial-contents
                         '(t   nil t   nil))
   :taste    (make-array 4
                         :initial-contents
                         '(t   t   t   nil))
   :movable nil))
```

```
;;; defining the left and right hand objects
```

object-1

```
(defvar *object-1*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(nil t   t   nil
                           t   nil nil t
                           t   nil nil t
                           nil t   t   nil))
   :tactile  (make-array 4
                         :initial-contents
                         '(nil nil t   t))
   :taste    (make-array 4
                         :initial-contents
                         '(nil t   t   nil))
   :movable t  :cycle 0))
```

object-2

```
(defvar *object-2*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(t   t   nil nil
                           nil t   t   nil
                           nil nil t   t
                           nil nil nil t))
   :tactile  (make-array 4
                         :initial-contents
                         '(nil t   nil nil))
   :taste    (make-array 4
                         :initial-contents
                         '(nil t   nil t  ))
   :movable t  :cycle 1))
```

```
;;; defining two objects used for microworld testing via the
;;; schema-microworld-test function in schema.lisp
```

generic-object

```
(defvar *generic-object*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(t   nil t   nil
                           nil t   nil t
                           t   nil t   nil
                           nil t   nil t))
   :tactile  (make-array 4
                         :initial-contents
                         '(nil t   nil nil))
   :taste    (make-array 4
                         :initial-contents
                         '(nil nil t   t  ))
   :movable t  :cycle 0))
```

immobile-generic-object

```
(defvar *immobile-generic-object*
  (make-world-object
   :visual   (make-array 16
                         :initial-contents
                         '(t   nil t   nil
                           nil t   nil t
                           t   nil t   nil
                           nil t   nil t))
   :tactile  (make-array 4
                         :initial-contents
                         '(nil t   nil nil))
   :taste    (make-array 4
                         :initial-contents
                         '(nil nil t   t  ))
   :movable nil))
```

```
;;; note on the body relative coordinate system:
;;; body relative coordinates are often used when referring to the
;;; center 3x3 area in the microworld
;;; the lower left corner of this area is defined as body relative
;;; coordinate (0,0) which corresponds to microspace coordinate (2,2)

;;; the position of the body in body relative coordinates is (1,-1)
;;; which corresponds to microspace coordinate (3,1)

(defconstant *body-x* 1)                                   *body-x*
(defconstant *body-y* -1)                                  *body-y*

;;; *clock* is incremented with each call to the function clock-tick
;;; *grip-expiration* is used by the clock-tick function to ensure
;;; that the hand is not closed for more than 3 consecutive time units

(defvar *clock* 0)                                          *clock*
(defvar *grip-expiration* -1)                     *grip-expiration*
```

```
;;;; *** start of primitive item definitions ***
;;;; the primitive items are implemented as functions which return
;;;; values which indicate the current state of a particular item
;;;; the value t indicates the item is currently on
;;;; the value nil indicates the item is currently off
;;;; collectively they indicate those states of the microworld
;;;; which the simulated robot is currently able to sense

;;; haptic-proprioceptive items hp00-hp22
;;; the variables *hp-x* and *hp-y* store the current body relative
;;; hand position - each ranges from 0-2

(defvar *hp-x* 1)                                              *hp-x*
(defvar *hp-y* 0)                                              *hp-y*

;;; hp-check returns t if x and y give the current hand position

(defmacro hp-check (x y)                                       hp-check
  (declare (fixnum x y))
  '(and
    (fix= *hp-x* ,x)
    (fix= *hp-y* ,y)))

;;; definition of the hp item functions via the hp-check macro

(defun hp00 ()                                                 hp00
  (hp-check 0 0))
(defun hp01 ()                                                 hp01
  (hp-check 0 1))
(defun hp02 ()                                                 hp02
  (hp-check 0 2))

(defun hp10 ()                                                 hp10
  (hp-check 1 0))
(defun hp11 ()                                                 hp11
  (hp-check 1 1))
(defun hp12 ()                                                 hp12
  (hp-check 1 2))

(defun hp20 ()                                                 hp20
  (hp-check 2 0))
(defun hp21 ()                                                 hp21
  (hp-check 2 1))
(defun hp22 ()                                                 hp22
  (hp-check 2 2))
```

```
;;; visual-proprioceptive items vp00-vp22
;;; the variables *vp-x* and *vp-y* store the current glance
;;; orientation - each ranges from 0-2
;;; they are microspace relative coordinates which designate the
;;; center of the 5x5 visual field

(defvar *vp-x* 1)                                              *vp-x*
(defvar *vp-y* 1)                                              *vp-y*

;;; vp-check returns t if x and y give the current glance orientation

(defmacro vp-check (x y)                                       vp-check
  (declare (fixnum x y))
  '(and
    (fix= *vp-x* ,x)
    (fix= *vp-y* ,y)))

;;; definition of the vp item functions via the vp-check macro

(defun vp00 ()                                                 vp00
  (vp-check 0 0))
(defun vp01 ()                                                 vp01
  (vp-check 0 1))
(defun vp02 ()                                                 vp02
  (vp-check 0 2))

(defun vp10 ()                                                 vp10
  (vp-check 1 0))
(defun vp11 ()                                                 vp11
  (vp-check 1 1))
(defun vp12 ()                                                 vp12
  (vp-check 1 2))

(defun vp20 ()                                                 vp20
  (vp-check 2 0))
(defun vp21 ()                                                 vp21
  (vp-check 2 1))
(defun vp22 ()                                                 vp22
  (vp-check 2 2))
```

```
;;; note on the glance relative coordinate system:
;;; glance relative coordinates are used when referring to the 5x5
;;; area centered around the current glance orientation
;;; the center of this area is defined as glance relative coordinate
;;; (2,2) with the lower left corner of this area defined as glance
;;; relative coordinate (0,0)
;;; the current glance orientation is stored as microspace relative
;;; coordinates in *vp-x* and *vp-y*

;;; coarse visual-field items vf00-vf44

;;; vf-check returns t if an object is at the given
;;; glance relative position - if empty, returns nil

(defmacro vf-check (x y)                                          vf-check
  (declare (fixnum x y))
  `(if (get-object (fix+ *vp-x* ,x) (fix+ *vp-y* ,y))
       t nil))

;;; definition of the vf item functions via the vf-check macro

(defun vf00 ()                                                    vf00
  (vf-check 0 0))
(defun vf01 ()                                                    vf01
  (vf-check 0 1))
(defun vf02 ()                                                    vf02
  (vf-check 0 2))
(defun vf03 ()                                                    vf03
  (vf-check 0 3))
(defun vf04 ()                                                    vf04
  (vf-check 0 4))

(defun vf10 ()                                                    vf10
  (vf-check 1 0))
(defun vf11 ()                                                    vf11
  (vf-check 1 1))
(defun vf12 ()                                                    vf12
  (vf-check 1 2))
(defun vf13 ()                                                    vf13
  (vf-check 1 3))
(defun vf14 ()                                                    vf14
  (vf-check 1 4))
```

;;; definition of the vf item functions via the vf–check macro

```
(defun vf20 ()                                                    vf20
  (vf-check 2 0))
(defun vf21 ()                                                    vf21
  (vf-check 2 1))
(defun vf22 ()                                                    vf22
  (vf-check 2 2))
(defun vf23 ()                                                    vf23
  (vf-check 2 3))
(defun vf24 ()                                                    vf24
  (vf-check 2 4))

(defun vf30 ()                                                    vf30
  (vf-check 3 0))
(defun vf31 ()                                                    vf31
  (vf-check 3 1))
(defun vf32 ()                                                    vf32
  (vf-check 3 2))
(defun vf33 ()                                                    vf33
  (vf-check 3 3))
(defun vf34 ()                                                    vf34
  (vf-check 3 4))

(defun vf40 ()                                                    vf40
  (vf-check 4 0))
(defun vf41 ()                                                    vf41
  (vf-check 4 1))
(defun vf42 ()                                                    vf42
  (vf-check 4 2))
(defun vf43 ()                                                    vf43
  (vf-check 4 3))
(defun vf44 ()                                                    vf44
  (vf-check 4 4))
```

```
;;; visual detail items
;;; fovf00-33, fovb00-33, fovl00-33, fovr00-33, fovx00-33

;;; each fov-check returns one of the visual details
;;; associated with the object found at the given foveal
;;; area - each detail is either t or nil
;;; note that the foveal areas are determined by the current
;;; microspace relative glance orientation

;;; note that the 2D items are kept internally in a 1D array so it is
;;; necessary to translate between the two representations when
;;; referencing the get-visual macro
;;; further note that this translation happens at compile time, not
;;; run time, so it doesn't slow down the execution of the code
```

fovf-check

```
(defmacro fovf-check (x y)
  (declare (fixnum x y))
  '(let ((foo (get-object (fix+ 2 *vp-x*) (fix+ 3 *vp-y*))))
     (and foo (get-visual foo ,(+ (* x 4) y)))))
```

fovb-check

```
(defmacro fovb-check (x y)
  (declare (fixnum x y))
  '(let ((foo (get-object (fix+ 2 *vp-x*) (fix1+ *vp-y*))))
     (and foo (get-visual foo ,(+ (* x 4) y)))))
```

fovl-check

```
(defmacro fovl-check (x y)
  (declare (fixnum x y))
  '(let ((foo (get-object (fix1+ *vp-x*) (fix+ 2 *vp-y*))))
     (and foo (get-visual foo ,(+ (* x 4) y)))))
```

fovr-check

```
(defmacro fovr-check (x y)
  (declare (fixnum x y))
  '(let ((foo (get-object (fix+ 3 *vp-x*) (fix+ 2 *vp-y*))))
     (and foo (get-visual foo ,(+ (* x 4) y)))))
```

fovx-check

```
(defmacro fovx-check (x y)
  (declare (fixnum x y))
  '(let ((foo (get-object (fix+ 2 *vp-x*) (fix+ 2 *vp-y*))))
     (and foo (get-visual foo ,(+ (* x 4) y)))))
```

;;; definition of the fovf item functions via the fovf–check macro

```
(defun fovf00 ()                                          fovf00
  (fovf-check 0 0))
(defun fovf01 ()                                          fovf01
  (fovf-check 0 1))
(defun fovf02 ()                                          fovf02
  (fovf-check 0 2))
(defun fovf03 ()                                          fovf03
  (fovf-check 0 3))
(defun fovf10 ()                                          fovf10
  (fovf-check 1 0))
(defun fovf11 ()                                          fovf11
  (fovf-check 1 1))
(defun fovf12 ()                                          fovf12
  (fovf-check 1 2))
(defun fovf13 ()                                          fovf13
  (fovf-check 1 3))
(defun fovf20 ()                                          fovf20
  (fovf-check 2 0))
(defun fovf21 ()                                          fovf21
  (fovf-check 2 1))
(defun fovf22 ()                                          fovf22
  (fovf-check 2 2))
(defun fovf23 ()                                          fovf23
  (fovf-check 2 3))
(defun fovf30 ()                                          fovf30
  (fovf-check 3 0))
(defun fovf31 ()                                          fovf31
  (fovf-check 3 1))
(defun fovf32 ()                                          fovf32
  (fovf-check 3 2))
(defun fovf33 ()                                          fovf33
  (fovf-check 3 3))
```

```
;;; definition of the fovb item functions via the fovb—check macro

(defun fovb00 ()                                                   fovb00
  (fovb—check 0 0))
(defun fovb01 ()                                                   fovb01
  (fovb—check 0 1))
(defun fovb02 ()                                                   fovb02
  (fovb—check 0 2))
(defun fovb03 ()                                                   fovb03
  (fovb—check 0 3))
(defun fovb10 ()                                                   fovb10
  (fovb—check 1 0))
(defun fovb11 ()                                                   fovb11
  (fovb—check 1 1))
(defun fovb12 ()                                                   fovb12
  (fovb—check 1 2))
(defun fovb13 ()                                                   fovb13
  (fovb—check 1 3))
(defun fovb20 ()                                                   fovb20
  (fovb—check 2 0))
(defun fovb21 ()                                                   fovb21
  (fovb—check 2 1))
(defun fovb22 ()                                                   fovb22
  (fovb—check 2 2))
(defun fovb23 ()                                                   fovb23
  (fovb—check 2 3))
(defun fovb30 ()                                                   fovb30
  (fovb—check 3 0))
(defun fovb31 ()                                                   fovb31
  (fovb—check 3 1))
(defun fovb32 ()                                                   fovb32
  (fovb—check 3 2))
(defun fovb33 ()                                                   fovb33
  (fovb—check 3 3))
```

;;; definition of the fovl item functions via the fovl–check macro

```
(defun fovl00 ()                                      fovl00
  (fovl-check 0 0))
(defun fovl01 ()                                      fovl01
  (fovl-check 0 1))
(defun fovl02 ()                                      fovl02
  (fovl-check 0 2))
(defun fovl03 ()                                      fovl03
  (fovl-check 0 3))
(defun fovl10 ()                                      fovl10
  (fovl-check 1 0))
(defun fovl11 ()                                      fovl11
  (fovl-check 1 1))
(defun fovl12 ()                                      fovl12
  (fovl-check 1 2))
(defun fovl13 ()                                      fovl13
  (fovl-check 1 3))
(defun fovl20 ()                                      fovl20
  (fovl-check 2 0))
(defun fovl21 ()                                      fovl21
  (fovl-check 2 1))
(defun fovl22 ()                                      fovl22
  (fovl-check 2 2))
(defun fovl23 ()                                      fovl23
  (fovl-check 2 3))
(defun fovl30 ()                                      fovl30
  (fovl-check 3 0))
(defun fovl31 ()                                      fovl31
  (fovl-check 3 1))
(defun fovl32 ()                                      fovl32
  (fovl-check 3 2))
(defun fovl33 ()                                      fovl33
  (fovl-check 3 3))
```

```
;;; definition of the fovr item functions via the fovr-check macro

(defun fovr00 ()                                          fovr00
  (fovr-check 0 0))
(defun fovr01 ()                                          fovr01
  (fovr-check 0 1))
(defun fovr02 ()                                          fovr02
  (fovr-check 0 2))
(defun fovr03 ()                                          fovr03
  (fovr-check 0 3))
(defun fovr10 ()                                          fovr10
  (fovr-check 1 0))
(defun fovr11 ()                                          fovr11
  (fovr-check 1 1))
(defun fovr12 ()                                          fovr12
  (fovr-check 1 2))
(defun fovr13 ()                                          fovr13
  (fovr-check 1 3))
(defun fovr20 ()                                          fovr20
  (fovr-check 2 0))
(defun fovr21 ()                                          fovr21
  (fovr-check 2 1))
(defun fovr22 ()                                          fovr22
  (fovr-check 2 2))
(defun fovr23 ()                                          fovr23
  (fovr-check 2 3))
(defun fovr30 ()                                          fovr30
  (fovr-check 3 0))
(defun fovr31 ()                                          fovr31
  (fovr-check 3 1))
(defun fovr32 ()                                          fovr32
  (fovr-check 3 2))
(defun fovr33 ()                                          fovr33
  (fovr-check 3 3))
```

```
;;; definition of the fovx item functions via the fovx-check macro

(defun fovx00 ()                                                     fovx00
  (fovx-check 0 0))
(defun fovx01 ()                                                     fovx01
  (fovx-check 0 1))
(defun fovx02 ()                                                     fovx02
  (fovx-check 0 2))
(defun fovx03 ()                                                     fovx03
  (fovx-check 0 3))
(defun fovx10 ()                                                     fovx10
  (fovx-check 1 0))
(defun fovx11 ()                                                     fovx11
  (fovx-check 1 1))
(defun fovx12 ()                                                     fovx12
  (fovx-check 1 2))
(defun fovx13 ()                                                     fovx13
  (fovx-check 1 3))
(defun fovx20 ()                                                     fovx20
  (fovx-check 2 0))
(defun fovx21 ()                                                     fovx21
  (fovx-check 2 1))
(defun fovx22 ()                                                     fovx22
  (fovx-check 2 2))
(defun fovx23 ()                                                     fovx23
  (fovx-check 2 3))
(defun fovx30 ()                                                     fovx30
  (fovx-check 3 0))
(defun fovx31 ()                                                     fovx31
  (fovx-check 3 1))
(defun fovx32 ()                                                     fovx32
  (fovx-check 3 2))
(defun fovx33 ()                                                     fovx33
  (fovx-check 3 3))
```

```
;;; coarse tactile items for each side of the hand
;;; tactf, tactb, tactr, tactl

;;; return t if an object is present, nil if not
;;; t or nil is returned explicitly so the user cannot 'accidentally'
;;; get an object structure

;;; note that the object positions checked are the microspace
;;; coordinates indicated by the current body relative position of the
;;; hand

(defun tactf ()                                                     tactf
  (if (get-object (fix+ 2 *hp-x*) (fix+ 3 *hp-y*))
      t nil))

(defun tactb ()                                                     tactb
  (if (get-object (fix+ 2 *hp-x*) (fix1+ *hp-y*))
      t nil))

(defun tactr ()                                                     tactr
  (if (get-object (fix+ 3 *hp-x*) (fix+ 2 *hp-y*))
      t nil))

(defun tactl ()                                                     tactl
  (if (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
      t nil))
```

```
;;; coarse tactile items for each side of the body
;;; bodyf, bodyb, bodyr, bodyl

;;; again, return t if an object is present, nil if not and
;;; t or nil is returned explicitly so the user cannot 'accidentally'
;;; get an object structure

;;; note that the object positions checked are the microspace
;;; coordinates indicated by the current body relative position of the
;;; body

(defun bodyf ()                                                    bodyf
  (if (get-object (fix+ 2 *body-x*) (fix+ 3 *body-y*))
      t nil))

(defun bodyb ()                                                    bodyb
  (if (get-object (fix+ 2 *body-x*) (fix1+ *body-y*))
      t nil))

(defun bodyr ()                                                    bodyr
  (if (get-object (fix+ 3 *body-x*) (fix+ 2 *body-y*))
      t nil))

(defun bodyl ()                                                    bodyl
  (if (get-object (fix1+ *body-x*) (fix+ 2 *body-y*))
      t nil))

;;; detailed tactile items for objects touching the left edge (i.e.
;;; "fingers") of the hand text0-3

;;; each detail is t or nil

;;; note that the position checked is the microspace coordinate
;;; indicated as the position to the left of the current body relative
;;; position of the hand

(defmacro text-check (x)                                           text-check
  (declare (fixnum x))
  '(let ((foo (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))))
     (if foo
         (get-tactile foo ,x)
      nil)))

(defun text0 ()                                                    text0
  (text-check 0))
(defun text1 ()                                                    text1
  (text-check 1))
(defun text2 ()                                                    text2
  (text-check 2))
(defun text3 ()                                                    text3
  (text-check 3))
```

```
;;; detailed taste items for objects touching the front (i.e. "mouth")
;;; of the body taste0-3

;;; each detail is t or nil

;;; note that the position checked is the microspace coordinate
;;; indicated as the position to the front of the current body relative
;;; position of the body

(defmacro taste-check (x)                                              taste-check
  (declare (fixnum x))
  `(let ((foo (get-object (fix+ 2  *body-x*) (fix+ 3 *body-y*))))
     (if foo
         (get-taste foo ,x)
       nil)))

(defun taste0 ()                                                       taste0
  (taste-check 0))
(defun taste1 ()                                                       taste1
  (taste-check 1))
(defun taste2 ()                                                       taste2
  (taste-check 2))
(defun taste3 ()                                                       taste3
  (taste-check 3))

;;; hand closed item hcl

(defvar *hcl* nil)                                                     *hcl*

(defun hcl ()                                                          hcl
  *hcl*)


;;; hand closed and grasping something item hgr

(defvar *hgr* nil)                                                     *hgr*

(defun hgr ()                                                          hgr
  *hgr*)


;;; items used solely for testing

(defun always-true ()                                                  always-true
  t)

(defun always-false ()                                                 always-false
  nil)
```

```
;;;; *** start of primitive action definitions ***
;;;; the primitive actions are implemented as functions which may or
;;;; may not change the microworld state
;;;; these functions do not return meaningful values

;;; hand motion functions handf, handb, handr, handl

;;; each function first checks the new position for conflicts
;;; (out of the 3x3 range, object already occupying new position)
;;; and if none, moves the hand by first moving any grasped object,
;;; and then moving the hand
;;; note that the destination square for both the hand and any grasped
;;; object must be empty for the hand movement to take place
;;; (however, when the hand is closed and grasping an object and moves
;;; left, the hand will occupy the former position of the grasped
;;; object so only the position left of the object needs to be empty,
;;; and similarly for moving right with a grasped object, the object
;;; will occupy the former position of the hand so only the position
;;; to the right of the hand needs to be empty)
```

handf

```
(defun handf ()
  (if (not (or (fix> *hp-y* 1)
               (get-object (fix+ 2 *hp-x*) (fix+ 3 *hp-y*))
               (and *hgr* (get-object (fix1+ *hp-x*) (fix+ 3 *hp-y*)))))
      (progn
        (if *hgr*
            (setf (get-object (fix1+ *hp-x*) (fix+ 3 *hp-y*))
                  (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
                  (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
                  nil))
        (setf (get-object (fix+ 2 *hp-x*) (fix+ 3 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              nil
              *hp-y* (fix1+ *hp-y*)))))
```

handb

```
(defun handb ()
  (if (not (or (fix< *hp-y* 1)
               (get-object (fix+ 2 *hp-x*) (fix1+ *hp-y*))
               (and *hgr* (get-object (fix1+ *hp-x*) (fix1+ *hp-y*)))))
      (progn
        (if *hgr*
            (setf (get-object (fix1+ *hp-x*) (fix1+ *hp-y*))
                  (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
                  (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
                  nil))
        (setf (get-object (fix+ 2 *hp-x*) (fix+ 1 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              nil
              *hp-y* (fix1- *hp-y*)))))
```

```
(defun handr ()                                                       handr
  (if (not (or (fix> *hp-x* 1)
               (get-object (fix+ 3 *hp-x*) (fix+ 2 *hp-y*))))
      (progn
        (setf (get-object (fix+ 3 *hp-x*) (fix+ 2 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)))
        (if *hgr*
            (setf
             (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
             (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
             (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))
             nil)
          (setf
           (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
           nil))
        (setf *hp-x* (fix1+ *hp-x*)))))

(defun handl ()                                                       handl
  (if (not (or (fix< *hp-x* 1)
               (and *hgr* (get-object *hp-x* (fix+ 2 *hp-y*)))
               (and (not *hgr*)
                    (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*)))))
      (progn
        (if *hgr*
            (setf (get-object *hp-x* (fix+ 2 *hp-y*))
                  (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))))
        (setf (get-object (fix+ 1 *hp-x*) (fix+ 2 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*))
              nil
              *hp-x* (fix1- *hp-x*)))))

;;; glance orientation functions eyef, eyeb, eyer, eyel

;;; each function merely checks to make sure that the glance
;;; orientation variable will still be in the range 0-2 inclusive
;;; after modification, and then performs the modification

(defun eyef ()                                                        eyef
  (if (fix< *vp-y* 2)
      (setf *vp-y* (fix1+ *vp-y*))))

(defun eyeb ()                                                        eyeb
  (if (fix> *vp-y* 0)
      (setf *vp-y* (fix1- *vp-y*))))

(defun eyer ()                                                        eyer
  (if (fix< *vp-x* 2)
      (setf *vp-x* (fix1+ *vp-x*))))

(defun eyel ()                                                        eyel
  (if (fix> *vp-x* 0)
      (setf *vp-x* (fix1- *vp-x*))))
```

```
;;; hand closing and opening functions grasp, ungrasp

;;; this function closes the hand, grasping any movable object
;;; touching the left edge of the hand (unless hand was already closed)
;;; the hand stays closed until 3 time units pass or until explicitly
;;; opened

(defun grasp ()                                                              grasp
  (if (not *hcl*)
      (progn
        (let ((object (get-object (fix1+ *hp-x*) (fix+ 2 *hp-y*))))
          (if (and object
                   (get-movable object))
              (setf *hgr* t)))
        (setf *hcl* t
              *grip-expiration* (fix+ 3 *clock*)))))

;;; this function opens the hand

(defun ungrasp ()                                                            ungrasp
  (setf *hcl* nil
        *hgr* nil
        *grip-expiration* -1))

;;; time-keeping function clock-tick

;;; each time unit should correspond to one primitive action taken
;;; this function should be called after each primitive action and
;;; before the schema mechanism takes the statistics for this time
;;; step - it returns the new value of *clock*
;;; the grip expires and the hand automatically opens after 3 time
;;; units from when it was first closed
;;; (i.e. if grasp is the action selected at time x, the hand will
;;; remain closed for time x+1, time x+2 and time x+3, unless
;;; explicitly opened during one of these time steps.  If it is not
;;; explicitly opened, after the action is executed for time x+3, the
;;; following call to clock-tick will open the hand and set the time
;;; equal to x+4.)
;;; on average, every 200 calls to this function randomly moves all of
;;; the non-grasped movable objects in the world (unless moving a
;;; particular object would result in a collision with another object)
;;; (1:200 -> 1,000:200,000 for additional randomness)
;;; the direction moved is indicated by the current cycle value for
;;; the object in question
;;; 0 indicates the next move is to the right, 1 down, 2 left, 3 up
;;; i.e. the objects move in a clockwise direction over time
```

```
(defun clock-tick ()
  (dotimes (x 7)
   (declare (fixnum x))
    (dotimes (y 7)
     (declare (fixnum y))
     (let ((foo (get-object x y)))
       (if (and foo
                (fix< (random 200000) 1000)
                (not
                 (and (fix= x (fix1+ *hp-x*))
                      (fix= y (fix+ 2 *hp-y*))
                      *hgr*))
                (get-movable foo))
           (let ((bar (get-cycle foo)))
             (declare (fixnum bar))
             (cond ((fix= bar 0)
                    (if (and (not (fix= x 8))
                             (not (get-object (fix1+ x) y)))
                        (setf (get-object (fix1+ x) y) foo
                              (get-object x y) nil
                              (get-cycle foo) 1)))
                   ((fix= bar 1)
                    (if (and (not (fix= y 0))
                             (not (get-object x (fix1- y))))
                        (setf (get-object x (fix1- y)) foo
                              (get-object x y) nil
                              (get-cycle foo) 2)))
                   ((fix= bar 2)
                    (if (and (not (fix= x 0))
                             (not (get-object (fix1- x) y)))
                        (setf (get-object (fix1- x) y) foo
                              (get-object x y) nil
                              (get-cycle foo) 3)))
                   ((fix= bar 3)
                    (if (and (not (fix= y 8))
                             (not (get-object x (fix1+ y))))
                        (setf (get-object x (fix1+ y)) foo
                              (get-object x y) nil
                              (get-cycle foo) 0)))))))))
  (if (and *hcl* (fix= *clock* *grip-expiration*))
      (setf *hcl* nil
            *hgr* nil
            *grip-expiration* -1))
  (setf *clock* (fix1+ *clock*)))
```

```
;;; random number generator seed saving function
;;; new-initial-random-state

;;; this function writes a new random-state seed
;;; to the disk file "state.dat"
;;; this seed will be used the next time the world is initialized
```

new-initial-random-state

```
(defun new-initial-random-state
  (&optional (filename "state.dat"))

  (with-open-file (out-file filename
                            :direction :output
                            :if-exists :supersede)
   (print (make-random-state t) out-file)
   (terpri out-file))
  'new-initial-random-state-written-to-disk)

;;; microworld initialization function init-world

;;; this function loads the current random seed from disk and resets
;;; all the world variables appropriately
;;; the hand is placed at body relative coordinate (1,0) which is
;;; microspace relative coordinate (3,2)
;;; the center of the visual field is oriented at body relative
;;; coordinate (1,1) which is microspace relative coordinate (3,3)
;;; the hand is left open
;;; the world is recreated from scratch, with the hand, body and two
;;; objects placed in the appropriate places, and the cycle values
;;; (used by clock-tick for randomly moving objects) reset
;;; finally, it resets and returns the value of *clock*
```

init-world

```
(defun init-world (&optional (filename "state.dat"))
  (with-open-file (in-file filename
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 0
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 1 5) *object-1* ; cycle 0 moves right
        (get-cycle *object-1*) 0
        (get-object 5 5) *object-2* ; cycle 1 moves down
        (get-cycle *object-2*) 1
        *clock* 0))
```

```
;;; these variations on the standard init-world are used to check the
;;; effect of fewer objects on the schemas built by the system
```

init-world-no-right-object

```
(defun init-world-no-right-object
  (&optional (filename "state.dat"))

  (with-open-file (in-file filename
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 0
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 1 5) *object-1* ; cycle 0 moves right
        (get-cycle *object-1*) 0
        *clock* 0))
```

init-world-no-left-object

```
(defun init-world-no-left-object
  (&optional (filename "state.dat"))

  (with-open-file (in-file filename
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 0
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 5 5) *object-2* ; cycle 1 moves down
        (get-cycle *object-2*) 1
        *clock* 0))
```

```
;;; these variations on the standard init-world are used to test the
;;; microworld via the schema-microworld-test function in schema.lisp
```

test1-init-world

```
(defun test1-init-world ()
  (with-open-file (in-file "state.dat"
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 0
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 3 3) *generic-object*
        (get-cycle *generic-object*) 0
        *clock* 0))
```

test2-init-world

```
(defun test2-init-world ()
  (with-open-file (in-file "state.dat"
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 0
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 3 3) *generic-object*
        (get-cycle *generic-object*) 0
        (get-object 2 4) *immobile-generic-object*
        *clock* 0))
```

test3-init-world

```
(defun test3-init-world ()
  (with-open-file (in-file "state.dat"
                           :direction :input
                           :if-does-not-exist :error)
   (setf *random-state* (read in-file nil nil nil)))
  (setf *grip-expiration* -1
        *hp-x* 1
        *hp-y* 1
        *vp-x* 1
        *vp-y* 1
        *hcl* nil
        *hgr* nil
        *world* (make-array '(7 7) :initial-element nil)
        (get-object (fix+ 2 *body-x*) (fix+ 2 *body-y*)) *body*
        (get-object (fix+ 2 *hp-x*) (fix+ 2 *hp-y*)) *hand*
        (get-object 2 3) *generic-object*
        (get-cycle *generic-object*) 0
        (get-object 3 2) *immobile-generic-object*
        *clock* 0))
```

A.2 Schema mechanism

```
;;; SCHEMA.LISP
;;; Schema mechanism implementation
;;; Part of the reimplementation of the CM2 implementation
;;; As defined and explained in
;;; "Made-up Minds: A Constructivist Approach to Artificial Intelligence"
;;; by Gary L. Drescher, as published by MIT Press 1991
;;; Originally published as a Ph.D. thesis September 1989
;;; Massachusetts Institute of Technology

;;; As part of the Computer Science Masters' thesis for
;;; Robert Ramstad, September 1992
;;; Massachusetts Institute of Technology
;;; Thesis Advisors: Professor Ronald Rivest
;;;   and Bruce Foster, Digital Equipment Corporation
;;; Copyright (c) 1992, Robert Ramstad, All Rights Reserved.
;;; The author hereby grants to MIT permission to reproduce and to
;;; distribute copies of this document in whole or in part.

;;; abbreviations used in this file:
;;; acc     = accessibility
;;; conj    = conjunction
;;; conjs   = conjunctions
;;; cons    = consistent
;;; consy   = consistency
;;; ext     = extended
;;; gd      = goal-directed
;;; lc      = local
;;; lcly    = locally
;;; neg     = negative
;;; pos     = positive
;;; stats   = statistics
;;; syn     = synthetic

;;; evaluate in interpreter before compiling if it is desired to see
;;; a vast amount of information regarding the compilation process
; (compiler-options :messages t :show-optimizations t :warnings t)

(in-package 'schema)

(use-package 'fix)
(use-package 'float)
```

```
;;; debugging output control

;;; each of these constants controls a certain number of related format
;;; statements -- if they are set to nil, the corresponding output is
;;; turned off by having the format statement macroexpand to nil
;;; if, however, the constant is set to t, the format statement
;;; expands to a form which sends the relevant output to *output-stream*
;;; NOTE: turning any of these off will result in a variable ARGS
;;; bound but not used error -- also note, however, that the variable
;;; ARGS is only used by the debugging printing control routines, so
;;; any error of this form can be safely ignored

(defvar *output-stream* t)                                       *output-stream*

(defconstant *schema-enabled* t)                                 *schema-enabled*
(defconstant *flag-array-enabled* nil)                           *flag-array-enabled*
(defconstant *state-array-enabled* nil)                          *state-array-enabled*
(defconstant *item-enabled* nil)                                 *item-enabled*
(defconstant *conj-enabled* nil)                                 *conj-enabled*
(defconstant *syn-item-enabled* nil)                             *syn-item-enabled*
(defconstant *applicable-enabled* nil)                           *applicable-enabled*
(defconstant *accessibility-enabled* nil)                        *accessibility-enabled*
(defconstant *activated-enabled* nil)                            *activated-enabled*
(defconstant *result-enabled* nil)                               *result-enabled*
(defconstant *reliability-enabled* nil)                          *reliability-enabled*
(defconstant *predicted-results-enabled* nil)                    *predicted-results-enabled*
(defconstant *ext-stats-enabled* nil)                            *ext-stats-enabled*
(defconstant *ext-conj-stats-enabled* nil)                       *ext-conj-stats-enabled*
(defconstant *show-items-enabled* nil)                           *show-items-enabled*
(defconstant *debug-enabled* nil)                                *debug-enabled*
```

```
;;; debugging output control

(defmacro main-format (&rest args)                                    main-format
  `(format *output-stream* ,@args))
(defmacro schema-format (&rest args)                                  schema-format
  (if *schema-enabled*
      `(format *output-stream* ,@args)))
(defmacro item-format (&rest args)                                    item-format
  (if *item-enabled*
      `(format *output-stream* ,@args)))
(defmacro conj-format (&rest args)                                    conj-format
  (if *conj-enabled*
      `(format *output-stream* ,@args)))
(defmacro syn-item-format (&rest args)                                syn-item-format
  (if *syn-item-enabled*
      `(format *output-stream* ,@args)))
(defmacro applicable-format (&rest args)                              applicable-format
  (if *applicable-enabled*
      `(format *output-stream* ,@args)))
(defmacro accessibility-format (&rest args)                           accessibility-format
  (if *accessibility-enabled*
      `(format *output-stream* ,@args)))
(defmacro activated-format (&rest args)                               activated-format
  (if *activated-enabled*
      `(format *output-stream* ,@args)))
(defmacro result-format (&rest args)                                  result-format
  (if *result-enabled*
      `(format *output-stream* ,@args)))
(defmacro reliability-format (&rest args)                             reliability-format
  (if *reliability-enabled*
      `(format *output-stream* ,@args)))
(defmacro predicted-results-format (&rest args)                       predicted-results-format
  (if *predicted-results-enabled*
      `(format *output-stream* ,@args)))
(defmacro ext-stats-format (&rest args)                               ext-stats-format
  (if *ext-stats-enabled*
      `(format *output-stream* ,@args)))
(defmacro ext-conj-stats-format (&rest args)                          ext-conj-stats-format
  (if *ext-conj-stats-enabled*
      `(format *output-stream* ,@args)))
(defmacro show-items-format (&rest args)                              show-items-format
  (if *show-items-enabled*
      `(format *output-stream* ,@args)))
(defmacro debug-format (&rest args)                                   debug-format
  (if *debug-enabled*
      `(format *output-stream* ,@args)))
```

```
;;; debugging output control

;;; these macros help to eliminate unnecessary code -- code which
;;; exists only to provide proper output -- if the relevant constant is
;;; nil, there is no reason to execute the code in question, as the
;;; system will not be outputting anything anyway

(defmacro applicable-if (&rest args)                        applicable-if
  (if *applicable-enabled*
      `(if ,@args)))
(defmacro activated-if (&rest args)                          activated-if
  (if *activated-enabled*
      `(if ,@args)))
(defmacro result-if (&rest args)                                result-if
  (if *result-enabled*
      `(if ,@args)))
(defmacro reliability-if (&rest args)                      reliability-if
  (if *reliability-enabled*
      `(if ,@args)))
(defmacro ext-stats-if (&rest args)                          ext-stats-if
  (if *ext-stats-enabled*
      `(if ,@args)))
(defmacro ext-conj-stats-if (&rest args)                ext-conj-stats-if
  (if *ext-conj-stats-enabled*
      `(if ,@args)))
```

```
;;; global variables

;;; from the original implementation, 90 % of 4K processors were used
;;; to store schemas so 3600 schemas is used as the maximum
;;; (the other 10% were used to store goal-directed actions)
;;; item array must be large enough to contain roughly 140 primitive
;;; items plus the synthetic items (estimated conservatively at 160)
;;; for a total of 300
;;; each conjunctive item must first exist as the result of a reliable
;;; schema -- max of 3600 -- estimate conservatively as 200
;;; *-maximum* -- maximum possible index for array
;;; *-number* -- next position to insert

(defvar *clock-tick* 0)                                  *clock-tick*

(defvar *schema-number* 0)                            *schema-number*
(defvar *item-number* 0)                                *item-number*
(defvar *conj-number* 0)                                *conj-number*
(defvar *syn-item-number* 0)                        *syn-item-number*
(defvar *action-number* 0)                            *action-number*

(proclaim '(fixnum *clock-tick* *schema-number*
                   *item-number* *syn-item-number*
                   *conj-number* *action-number*))

(defconstant *schema-maximum* 3600)                  *schema-maximum*
(defconstant *item-maximum* 500)                       *item-maximum*
(defconstant *conj-maximum* 200)                       *conj-maximum*
(defconstant *syn-item-maximum* 350)               *syn-item-maximum*
(defconstant *action-maximum* 25)                    *action-maximum*
```

```
;;; flag datatype

;;; datatype used to indicate true or false for various things about
;;; schemas, goal-directed actions and synthetic items
;;; used repeatedly in all structures with compressed data
;;; uses fixnum math and implemented with macros should be efficient
;;; not separately tested, but tested thoroughly via item and schema
;;; testing
;;; true represented as 1, false represented as 0
;;; (implemented as bits to save memory, see flag-record below)

;;; notes on functions:
;;; flag-parse
;;;   takes t/nil returns true/false flag
;;; flag-unparse
;;;   takes true/false flag returns "+"/"-"

(defmacro make-flag-true () '(the fixnum 1))                           make-flag-true
(defmacro make-flag-false () '(the fixnum 0))                         make-flag-false
(defmacro flag-eq (x y) '(fix= ,x ,y))                                        flag-eq
(defmacro flag-truep (x) '(flag-eq (make-flag-true) ,x))                   flag-truep
(defmacro flag-falsep (x) '(flag-eq (make-flag-false) ,x))                flag-falsep
(defmacro flag-inverse (x)                                               flag-inverse
  '(if (flag-truep ,x)
       (make-flag-false)
     (make-flag-true)))
(defmacro flag-parse (x)                                                   flag-parse
  '(if ,x
       (make-flag-true)
     (make-flag-false)))
(defmacro flag-unparse (x)                                               flag-unparse
  '(if (flag-truep ,x)
       "+"
     "-"))
```

```
;;; flag-record datatype

;;; simple way of packing up to 24 flags together into one fixnum
;;; flag-record-get-initial can be called to determine what the
;;; value returned by make-flag-record (a flag record which initially
;;; has all values set false) should be

;;; notes on functions:
;;; flag-record-get-initial
;;;   returns the value which should be used by make-flag-record
```

make-flag-record

```
(defmacro make-flag-record () '(the fixnum 0))
```

flag-record-get-flag

```
(defmacro flag-record-get-flag (flag-record position)
  '(ldb (byte 1 ,position) ,flag-record))
```

flag-record-get-initial

```
(defun flag-record-get-initial ()
  (let ((foo 0))
    (dotimes
     (x 24)
     (setf (flag-record-get-flag foo x) (make-flag-false)))
    foo))
```

```
;;; flag-array datatype

;;; fairly simple way of keeping more than 24 flags and accessing them
;;; in a transparent manner

;;; notes on functions:
;;; flag-array-get-flag
;;;   takes array and flag-index and returns flag
;;; flag-array-get-flag-specific
;;;   takes array, array-index and record-position and returns flag
;;; note: these two accessors support different ways of getting the
;;; same information, -specific is faster in those cases where the two
;;; values are already computed
;;; flag-array-ior
;;;   this function does an inclusive or between the source array and
;;;   destination array and places the result in the destination array
;;;   it is a big win in certain situations and some code has been
;;;   written to take advantage of it
;;; flag-array-included-p
;;;   returns t if all the on flags (1s) in the slave-array are also
;;;   on in the master-array
```

make-flag-array

```
(defmacro make-flag-array (max-index)
  `(make-array ,max-index
               :initial-element (make-flag-record)
               :element-type 'fixnum))
```

flag-array-get-flag

```
(defmacro flag-array-get-flag (array flag-index)
  `(flag-record-get-flag (aref ,array (floor ,flag-index 24))
                         (rem ,flag-index 24)))
```

flag-array-get-flag-specific

```
(defmacro flag-array-get-flag-specific
  (array array-index record-position)

  `(flag-record-get-flag (aref ,array ,array-index) ,record-position))
```

flag-array-ior

```
(defmacro flag-array-ior
  (source-array dest-array max-index)

  `(dotimes
    (x ,max-index)
    (setf (aref ,dest-array x) (logior (aref ,dest-array x)
                                       (aref ,source-array x)))))
```

;;; flag–array datatype functions continued

flag–array–included–p

```
(defmacro flag-array-included-p
  (master-array slave-array max-index)

  `(dotimes
    (x ,max-index t)
    (let ((slave-entry (aref ,slave-array x)))
      (if (fix/= 0 slave-entry)
          (if (fix/= slave-entry
                     (logand (aref ,master-array x) slave-entry))
              (return nil))))))
```

flag–array–clear

```
(defmacro flag-array-clear
  (array max-index)

  `(dotimes
    (x ,max-index)
    (setf (aref ,array x) (make-flag-record))))
```

flag–array–copy

```
(defmacro flag-array-copy
  (source-array dest-array max-index)

  `(dotimes
    (x ,max-index)
    (setf (aref ,dest-array x) (aref ,source-array x))))
```

print–flag–array

```
(defmacro print-flag-array (&rest args)
  (if *flag-array-enabled*
      `(print-flag-array-enabled ,@args)))
```

print–flag–array–enabled

```
(defmacro print-flag-array-enabled
  (array max-index)

  `(dotimes
    (x ,max-index)
    (dotimes
     (y 24)
     (format *output-stream* "~A "
             (flag-unparse
              (flag-array-get-flag-specific ,array x y))))
    (format *output-stream* "~%")))
```

```
;;; state datatype

;;; used to indicate state of primitive/conjunctive/synthetic items
;;; can be on, off or unknown
;;; uses fixnum math and implemented with macros should be efficient
;;; not separately tested, but tested thoroughly as used in many
;;; different functions
;;; note that all of these functions depend on representation
;;; on represented as 2, off as 1, unknown as 0

;;; notes on functions:
;;; make-state-both
;;;   this is used by the implementation as a placeholder when an item
;;;   could be either on or off especially for some esoteric uses of
;;;   state-array-ior
;;; state-inverse
;;;   takes on/off state returns off/on
;;;   does NOT work for unknown states
;;; state-parse
;;;   takes t/nil returns on/off state
;;; state-unparse
;;;   takes on/off/unknown state returns ”*”/”.”/”?”

(defmacro make-state-on () '(the fixnum 2))                    make-state-on
(defmacro make-state-off () '(the fixnum 1))                   make-state-off
(defmacro make-state-unknown () '(the fixnum 0))           make-state-unknown
(defmacro make-state-both () '(the fixnum 3))                make-state-both
(defmacro state-eq (state1 state2)                                   state-eq
  '(fix= ,state1 ,state2))
(defmacro state-noteq (state1 state2)                             state-noteq
  '(fix/= ,state1 ,state2))
(defmacro state-on-p (state)                                       state-on-p
  '(state-eq (make-state-on) ,state))
(defmacro state-off-p (state)                                     state-off-p
  '(state-eq (make-state-off) ,state))
(defmacro state-unknown-p (state)                             state-unknown-p
  '(state-eq (make-state-unknown) ,state))
(defmacro state-inverse (state)                                 state-inverse
  '(lognot ,state))
(defmacro state-parse (condition)                                 state-parse
  '(if ,condition
       (make-state-on)
     (make-state-off)))
(defun state-unparse (state)                                    state-unparse
  (cond ((state-on-p state) "*")
        ((state-off-p state) ".")
        (t "?")))
```

```
;;; state-record datatype

;;; simple way of packing up to 12 states together into one fixnum
;;; state-record-get-initial can be called to determine what the value
;;; returned by make-state-record (a state record which initially has
;;; all states set unknown) should be
```

make-state-record

```
(defmacro make-state-record () '(the fixnum 0))
```

state-record-get-state

371

```
(defmacro state-record-get-state (number position)
  '(ldb (byte 2 (fix* ,position 2)) ,number))
```

state-record-get-initial

```
(defun state-record-get-initial ()
  (let ((foo 0))
    (dotimes
     (x 12)
     (setf (state-record-get-state foo x) (make-state-unknown)))
    foo))
```

```
;;; state-array datatype

;;; used to keep item conjunctions within schema, in conj array
;;; and for various other purposes
;;; highly dependent on representation of state datatype
;;; represented as an array of state-records where each state-record
;;; can hold 12 states

;;; notes on functions:
;;; state-array-get-state
;;;   takes array and state-index and returns state
;;; state-array-get-state-specific
;;;   takes array, array-index and record-position and returns state
;;; note: these two accessors support different ways of getting the
;;; same information, -specific is faster in those cases where the two
;;; values are already computed
;;; state-array-copy-pos-flag
;;;   this function takes a source state-array and a destination
;;;   flag-array, wherever the state-array has an on state, the
;;;   corresponding flag array position gets set true
;;; state-array-copy-neg-flag
;;;   similar function to state-array-copy-pos-flag exccpt the
;;;   destination flag-array has flag positions set true wherever the
;;;   source state-array has an off state
;;; state-array-ior
;;;   this function does a bitwise inclusive or between the source
;;;   array and destination array and places the result in the
;;;   destination array
;;;   it is a big win in certain situations and some code has been
;;;   written to take advantage of it
;;; state-array-included-p
;;;   this function is used to check if a if a context/result is
;;;   satisfied, it takes two state-arrays and indicates if the second
;;;   array is included in the first, it assumes representation of
;;;   states as 00 is unknown, 10 is on, 01 is off
;;;   given the state of the microworld as a state-array s, and a
;;;   context state-array c, the context is satisfied if (and s c) = c
;;;   on the other hand, if some context bit is not matched by the
;;;   state of the microworld, then the result of (and s c) will
;;;   differ from c in that bit position
;;; state-array-get-print-name
;;;   given a state-array, this function returns a human-readable
;;;   string which corresponds to the state-array if it were a
;;;   conjunction of items, where having a non-unknown state in a
;;;   given position indicates that that particular item is included
;;;   in the conj (positive if on, negative if off)
```

;;; state-array datatype functions

make-state-array

```
(defmacro make-state-array (max-index)
  '(make-array ,max-index
               :initial-element (make-state-record)
               :element-type 'fixnum))
```

state-array-get-state

```
(defmacro state-array-get-state (array state-index)
  '(state-record-get-state
    (aref ,array (floor ,state-index 12))
    (rem ,state-index 12)))
```

state-array-get-state-specific

```
(defmacro state-array-get-state-specific
  (array array-index record-position)

  '(state-record-get-state
    (aref ,array ,array-index)
    ,record-position))
```

state-array-copy-pos-flag

```
(defmacro state-array-copy-pos-flag
  (source-array dest-array max-index)

  '(let ((count 0))
     (dotimes
      (x ,max-index)
      (let ((num (aref ,source-array x)))
        (dotimes
         (y 12)
         (if (state-on-p (state-record-get-state num y))
             (setf (flag-array-get-flag ,dest-array count)
                   (make-flag-true)))
         (setf count (fix1+ count)))))))
```

state-array-copy-neg-flag

```
(defmacro state-array-copy-neg-flag
  (source-array dest-array max-index)

  '(let ((count 0))
     (dotimes
      (x ,max-index)
      (let ((num (aref ,source-array x)))
        (dotimes
         (y 12)
         (if (state-off-p (state-record-get-state num y))
             (setf (flag-array-get-flag ,dest-array count)
                   (make-flag-true)))
         (setf count (fix1+ count)))))))
```

;;; state-array datatype functions continued

```
(defmacro state-array-ior                                  state-array-ior
  (source-array dest-array max-index)

  `(dotimes
    (x ,max-index)
    (setf (aref ,dest-array x) (logior (aref ,dest-array x)
                                       (aref ,source-array x)))))

(defmacro state-array-included-p                      state-array-included-p
  (master-array slave-array max-index)

  `(dotimes
    (x ,max-index t)
    (let ((slave-enty (aref ,slave-array x)))
      (if (fix/= 0 slave-enty)
          (if (fix/= slave-enty
                     (logand (aref ,master-array x) slave-enty))
              (return nil))))))

(defmacro state-array-get-print-name             state-array-get-print-name
  (array)

  `(let ((result ""))
     (dotimes
      (x *item-number* (string-right-trim "&" result))
      (let ((state (state-array-get-state ,array x)))
        (if (not (state-unknown-p state))
            (setq result
                  (concatenate 'string
                               result
                               (if (state-off-p state)
                               "-")
                               (item-print-name (get-item x))
                               "&")))))))

(defmacro state-array-clear (array max-index)              state-array-clear
  `(dotimes
    (x ,max-index)
    (setf (aref ,array x) (make-state-record))))

(defmacro state-array-copy                                 state-array-copy
  (source-array dest-array max-index)

  `(dotimes
    (x ,max-index)
    (setf (aref ,dest-array x) (aref ,source-array x))))
```

;;; state-array datatype functions continued

print-state-array

```
(defmacro print-state-array (&rest args)
  (if *state-array-enabled*
      '(print-state-array-enabled ,@args)))
```

print-state-array-enabled

```
(defmacro print-state-array-enabled
  (array max-index)

  '(dotimes
    (x ,max-index)
    (dotimes
     (y 12)
     (format *output-stream* "~A "
            (state-unparse
             (state-array-get-state ,array (fix+ y (fix* x 12))))))
    (format *output-stream* "~%")))
```

microworld-state

get-microworld-state

```
(defvar *microworld-state* (make-state-array 25))
(defmacro get-microworld-state (index)
  '(state-array-get-state *microworld-state* ,index))
```

```
;;; rate datatype

;;; used to keep track of how often something occurs
;;; kept as a ratio of fixnums
;;; every time the rate is updated, the denominator is incremented
;;; the numerator is incremented iff the occurrence happened
;;; the rate is therefore a fraction giving the rate of occurrence
;;; (i.e. 3/8 means that the situation in question occurred 3 times out
;;; of 8 times total)
;;; to avoid overflow, the maximum value for either denominator or
;;;   numerator is (2^5) - 1 = 31 -- if above this, both are right
;;;   shifted one bit, preserving the ratio (more or less)
;;; a rate therefore consumes 10 bits total, where *rate-bits* gives
;;;   the number of bits for either the numerator or denominator
;;; the initial rate is 0/1, with denominator LSB in the data, this
;;;   simplifies to 1
;;; note that all rate functions take an offset, which can be used to
;;;   pack two rates into a single fixnum, or pack a rate plus some
;;;   flags and/or states into a single fixnum
;;; to compare the magnitude of two rates, think of them as fractions,
;;;   one simply compares numerator1 * denominator2 with numerator2 *
;;;   denominator1 (it should never be possible to have a rate of 0/0,
;;;   as the initial rate is 0/1 and calls to rate-update always
;;;   increment the denominator, so this method always works)
;;; rate-high-p uses this method, returns t if rate is greater than 7/8

(defconstant *rate-bits* 5)                                   *rate-bits*

(defmacro make-rate ()                                        make-rate
  '1)

(defmacro rate-denominator (data offset)                      rate-denominator
  '(ldb (byte *rate-bits* ,offset) ,data))

(defmacro rate-numerator (data offset)                        rate-numerator
  '(ldb (byte *rate-bits* (fix+ *rate-bits* ,offset)) ,data))

(defun rate-unparse (data offset)                             rate-unparse
  (format nil "~D/~D"
          (rate-numerator data offset)
          (rate-denominator data offset)))
```

;;; rate datatype functions continued

rate-update

```
(defmacro rate-update (data offset occurred)
  '(if ,occurred
       (rate-increment-both ,data ,offset)
     (rate-increment-denominator ,data ,offset)))
```

rate-increment-denominator

```
(defmacro rate-increment-denominator
  (data offset)

  '(let ((value (fix1+ (rate-denominator ,data ,offset))))
     (if (logbitp *rate-bits* value)
         (setf (rate-denominator ,data ,offset)
               (fixrsh value)
               (rate-numerator ,data ,offset)
               (fixrsh (rate-numerator ,data ,offset)))
       (setf (rate-denominator ,data ,offset) value))))
```

rate-increment-both

```
(defmacro rate-increment-both
  (data offset)

  '(let ((num-value (fix1+ (rate-numerator ,data ,offset)))
         (den-value (fix1+ (rate-denominator ,data ,offset))))
     (if (logbitp *rate-bits* den-value)
         (setf (rate-numerator ,data ,offset)
               (fixrsh num-value)
               (rate-denominator ,data ,offset)
               (fixrsh den-value))
       (setf (rate-numerator ,data ,offset) num-value
             (rate-denominator ,data ,offset) den-value))))
```

rate<

```
(defmacro rate< (data1 offset1 data2 offset2)
  '(fix< (fix* (rate-numerator ,data1 ,offset1)
               (rate-denominator ,data2 ,offset2))
         (fix* (rate-numerator ,data2 ,offset2)
               (rate-denominator ,data1 ,offset1))))
```

rate-high-p

```
(defmacro rate-high-p (data offset)
  '(fix> (fix* (rate-numerator ,data ,offset) 8)
         (fix* (rate-denominator ,data ,offset) 7)))
```

```
;;; datatype weighted-rate

;;; used to keep track of how often something occurs, weighted towards
;;; more recent occurrences
;;; kept as a short-float which indicates the probability of something
;;; occurring (could be kept as a ratio of fixnums)
```

make-weighted-rate

```
(defmacro make-weighted-rate ()
  `0.0)
```

weighted-rate*

```
(defmacro weighted-rate* (num1 num2)
  `(float* ,num1 ,num2))
```

weighted-rate-decrement

```
(defmacro weighted-rate-decrement (num)
  `(setf ,num (float* .7 ,num)))
```

weighted-rate-increment

```
(defmacro weighted-rate-increment (num)
  `(setf ,num (float+ .3 (float* .7 ,num))))
```

weighted-rate-update

```
(defmacro weighted-rate-update (num occurred)
  `(if ,occurred
       (weighted-rate-increment ,num)
     (weighted-rate-decrement ,num)))
```

weighted-rate-high
weighted-rate-medium
weighted-rate-low

```
(defconstant *weighted-rate-high* .875)
(defconstant *weighted-rate-medium* .75)
(defconstant *weighted-rate-low* .40)
```

weighted-rate-high-p
weighted-rate-medium-p
weighted-rate-low-p

```
(defmacro weighted-rate-high-p (num)
  `(float> ,num *weighted-rate-high*))
(defmacro weighted-rate-medium-p (num)
  `(float> ,num *weighted-rate-medium*))
(defmacro weighted-rate-low-p (num)
  `(float< ,num *weighted-rate-low*))
```

```
;;; datatype average

;;; used to keep fixnum average durations for synthetic items
;;; stored as value and number of samples
```

average
661

```
(defstruct
  (average
;   (:print-function print-average)
   )
  (value 0 :type fixnum)
  (sample 0 :type fixnum))
```

```
;; results of the +,*,/ operations not guaranteed to be fixnums
```

average-update
670

```
(defmacro average-update (average new-value)
  '(setf (average-value ,average)
         (floor
          (/ (+ (the fixnum ,new-value)
                (* (the fixnum (average-value ,average))
                   (the fixnum (average-sample ,average))))
             (setf (average-sample ,average)
                   (fix1+ (average-sample ,average)))))))
```

```
;;; datatype counter

;;; used to keep correlations
;;; the maximum value is 15, and value is always a positive integer
;;; the positive flag is used to indicate if the value is positive or
;;;   negative
;;; toggle is used for purposes of tabulating frequency, as follows:
;;; to determine if a correlation exists between an initial and
;;; final condition, go through a number of trials
;;; if toggle is true, and the given condition holds, and the final
;;;   condition obtains, then increment the value
;;; if toggle is false, and the final condition obtains, then
;;;   decrement the value
;;; alternate trials (i.e. toggle toggle after each trial) -- as the
;;; decrement is larger than the increment, if the frequency with
;;; which conditions -> final exceeds the frequency in which final
;;; obtains (without checking any conditions) by more than the ratio
;;; of decrement to increment, then the value will eventually max out
;;; at 15 (note: this scheme always exerts pressure towards zero)
;;; counters are initialized toggle (bit 0) and positive (bit 1) set
;;;   on (both are flags, so on=1 and off=0) and value 0 (bits 2-5)
;;; each function takes an offset to support putting multiple counters
;;;   or mixed counters and flags/states/rates in the same fixnum

(defconstant *counter-bits* 6)                              *counter-bits*
(defconstant *counter-maximum* 15)                       *counter-maximum*

(defmacro make-counter ()                                     make-counter
  '3)

(defmacro counter (data offset)                                    counter
  `(ldb (byte 6 ,offset) ,data))
(defmacro counter-toggle (data offset)                      counter-toggle
  `(ldb (byte 1 ,offset) ,data))
(defmacro counter-pos (data offset)                            counter-pos
  `(ldb (byte 1 (fix1+ ,offset)) ,data))
(defmacro counter-value (data offset)                        counter-value
  `(ldb (byte 4 (fix+ 2 ,offset)) ,data))
```

```
;;; datatype counter functions continued
```

counter-increment-value

```
(defmacro counter-increment-value (data offset)
  `(let ((value (counter-value ,data ,offset)))
     (if (fix< value 13)
         (setf (counter-value ,data ,offset) (fix+ value 2))
       (if (fix/= value 15)
           (setf (counter-value ,data ,offset) 15)))))
```

counter-decrement-value

```
(defmacro counter-decrement-value (data offset)
  `(let ((value (counter-value ,data ,offset)))
     (cond
      ;; 3 or more, simply subtract three
      ((fix> value 2)
       (setf (counter-value ,data ,offset) (fix- value 3)))
      ;; if zero, become -2 (1 correlation in the negative direction)
      ((fix= value 0)
       (setf (counter-pos ,data ,offset) (make-flag-false)
             (counter-value ,data ,offset) 2))
      ;; otherwise, if 1 or 2, become 0 (positive sign)
      (t
       (setf (counter-pos ,data ,offset) (make-flag-true)
             (counter-value ,data ,offset) 0)))))
```

counter-modify-value

```
(defmacro counter-modify-value (data offset activated)
  `(if (flag-truep (counter-pos ,data ,offset))
       (if (flag-truep ,activated)
           (counter-increment-value ,data ,offset)
         (counter-decrement-value ,data ,offset))
     (if (flag-truep ,activated)
         (counter-decrement-value ,data ,offset)
       (counter-increment-value ,data ,offset))))
```

counter-togglep

```
(defmacro counter-togglep (data offset)
  `(flag-truep (counter-toggle ,data ,offset)))
```

counter-toggle-toggle

```
(defmacro counter-toggle-toggle (data offset)
  `(setf (counter-toggle ,data ,offset)
         (flag-inverse (counter-toggle ,data ,offset))))
```

counter-unparse

```
(defmacro counter-unparse (data offset)
  (declare (fixnum data offset))
  `(format nil "~A~2D~A"
           (flag-unparse (counter-pos ,data ,offset))
           (counter-value ,data ,offset)
           (if (flag-truep (counter-toggle ,data ,offset))
               "W"
             "O")))
```

```
;;; counter-record datatype

;;; simple way of packing 4 counters together into one fixnum
;;; counter-record-get-initial can be called to determine what the
;;; value returned by make-counter-record (a counter record which
;;; initially has all counters set value 0 positive and toggle both
;;; on) should be

(defconstant *counter-record-max-offset* 18)  *counter-record-max-offset*

(defmacro counter-record-offset (position)            counter-record-offset
  `(fix* 6 ,position))

(defmacro make-counter-record ()                      make-counter-record
  '798915)

(defun counter-record-get-initial ()             counter-record-get-initial
  (let ((record 0))
    (setf (counter record (counter-record-offset 0)) (make-counter))
    (setf (counter record (counter-record-offset 1)) (make-counter))
    (setf (counter record (counter-record-offset 2)) (make-counter))
    (setf (counter record (counter-record-offset 3)) (make-counter))
    record))

(defun counter-record-unparse (record)             counter-record-unparse
  (format nil "~A  ~A  ~A  ~A"
          (counter-unparse record (counter-record-offset 0))
          (counter-unparse record (counter-record-offset 1))
          (counter-unparse record (counter-record-offset 2))
          (counter-unparse record (counter-record-offset 3))))
```

```
;;; datatype counter-array

;;; used to keep numerous counters
;;; perhaps more appropriately called counter-record-array

(defmacro make-counter-array (number)                        make-counter-array
  '(make-array ,number
               :initial-element (make-counter-record)
               :element-type 'fixnum))

(defmacro get-counter-array-index (index)                    get-counter-array-index
  '(floor ,index 4))
(defmacro get-counter-record-position (index)                get-counter-record-position
  '(rem ,index 4))
(defmacro get-counter-record (array index)                   get-counter-record
  '(aref ,array ,index))

(defmacro counter-array-modify-value                         counter-array-modify-value
  (array index offset occurred)

  '(counter-modify-value
    (get-counter-record ,array ,index) ,offset ,occurred))

(defmacro counter-array-value                                counter-array-value
  (array index offset)

  '(counter-value
    (get-counter-record ,array ,index) ,offset))

(defmacro counter-array-pos                                  counter-array-pos
  (array index offset)

  '(counter-pos
    (get-counter-record ,array ,index) ,offset))

(defmacro counter-array-toggle-toggle                        counter-array-toggle-toggle
  (array index offset)

  '(counter-toggle-toggle
    (get-counter-record ,array ,index) ,offset))

(defmacro counter-array-toggle                               counter-array-toggle
  (array index offset)

  '(counter-toggle
    (get-counter-record ,array ,index) ,offset))

(defmacro counter-unparse-from-array                         counter-unparse-from-array
  (array index offset)

  '(counter-unparse
    (get-counter-record ,array ,index) ,offset))
```

```
;;; schema datatype

;;; used to store all schemas created by the system
;;; context/action/result "defines" schema (and are used to construct
;;;   its print-name)
;;; schemas are *never* duplicated
;;; schema claims if context met, taking action yields specified
;;;   result (with given reliability factor)
;;; many statistics concern activation, taking the action of a schema
;;;   when its context is satisfied
;;; if syn-item is true, reifier gives the syn-item index
;;;   and first-tick stores a clock tick time for use in updating the
;;;   on and off-durations for the reifying synthetic item
;;; otherwise no synthetic item has been created for this schema
;;; reliability indicates how often the result obtains when the schema
;;;   is activated (i.e. how often the schema succeeds) and is biased
;;;   towards more recent trials (by using the weighted-rate functions)
;;; parent gives the schema index of the parent schema
;;;   -1 if there is no parent (i.e. the schema is blank)
;;; context/result-children are state-arrays which have a given
;;;   position on/off if this schema has spun off a context/result
;;;   schema with that particular item positively/negatively included
;;; result-conj-children is a flag array which has a given
;;;   position true/false if this schema has spun off a result schema
;;;   with that particular conjunction positively included
;;; the three children structures together keep the mechanism from
;;;   accidentally duplicating an existing schema and also
;;;   context-children structure supports deferral to a more specific
;;;   schema
;;; if result non-empty extended-context looks for items ON or OFF
;;;   before action which affect probability of success when activated
;;;   keeps track of both positive and negative correlations for
;;;     *items* only (not conjunctions)
;;;   alternate between trials with and without activation of schema
;;;     using the compressed count method described in the thesis
;;;   if goal-directed-action schema, also keeps same correlations in
;;;     extended-context-post but with respect to value of items after
;;;     the goal-directed action is executed (currently not implemented)
;;; if result empty, extended-result looks for item transitions which
;;;   appear to be predicted by the activation of the schema
;;;   keeps track of both positive and negative transition
;;;     correlations for primitive items as well as conjunctions
;;;     representing contexts of reliable schemas
;;;   alternate between trials with and without activation of schema
;;;     using the compressed count method described in the thesis
;;; schema datatype

;;; the data slot holds many important status bits in a compressed
;;;   form
;;; the applicable, overridden and activated flags are used by the
;;;   toplevel to keep track of the status of the various schemas
;;; succeeded-last is a flag used to help keep lc-consistency
;;;   indicates that the schema succeeded last time it was activated
;;; marked is a flag used by the update accessibility routines to keep
;;;   track of which schemas have already been visited by the
;;;   algorithm
;;; lc-consy is a rate used to keep probability of successful
;;;   activation given that the previous activation was successful
;;; if lc-consy is high, the lcly-cons flag is set
;;;   true, a synthetic item is created for the schema, and the schema
;;;   mechanism begins to keep track of the on and off duration for
;;;   the synthetic item
```

schema

```
;;; schema datatype

(defstruct
 (schema
  (:constructor make-schema-internal
                (action-item action))
  (:print-function print-schema))
 (print-name "" :type string)
 (data 32768 :type fixnum)

 ;; empty context if context-empty is true
 ;; single item in context if context-single is true
 ;;   (item is in context-array)
 ;; multiple items in context if context-single is false
 ;; context is *always* in context-array
 ;; can also be found as conjunction if context-conj is true
 ;;   (conj index is in context-item)
 (context-array
  (make-state-array 25)
  :type (vector fixnum 25))
 (context-item -1 :type fixnum)

 ;; empty result if result-empty is true
 ;; result is conjunction if result-conj is true
 ;;   conj index in result-item
 ;; otherwise item index is in result-item
 ;; item is negated if result-negated is true
 ;; (conjunctions are never negated)
 (result-item -1 :type fixnum)

 ;; action is goal-directed if action-gd is true
 ;;   goal-directed-action index in action-item (not yet supported)
 ;; otherwise action index in action-item
 ;; action is kept for rapid lookup, could easily be deleted
 (action #'microworld:always-true :type compiled-function)
 (action-item -1 :type fixnum)

 (reifier -1 :type fixnum)
 (first-tick -1 :type fixnum)
 (reliability 0.0 :type short-float)
 (parent -1 :type fixnum)
 (context-children (make-state-array 25) :type (vector fixnum 25))
 (result-children  (make-state-array 25) :type (vector fixnum 25))
 (result-conj-children  (make-flag-array 10) :type (vector fixnum 10))
 (extended-context
  (make-counter-array 75) :type (vector fixnum 75))
 (extended-context-post
  (make-counter-array 75) :type (vector fixnum 75))
 (extended-result-pos
  (make-counter-array 75) :type (vector fixnum 75))
 (extended-result-neg
  (make-counter-array 75) :type (vector fixnum 75))
 (extended-result-conj-pos
  (make-counter-array 50) :type (vector fixnum 50)))
```

```
;;; schema datatype functions

;;; data contains important status bits
;;; pos length
;;; 0   1  context-empty
;;; 1   1  context-single
;;; 2   1  context-conj
;;; 3   1  result-empty
;;; 4   1  result-conj
;;; 5   1  result-negated
;;; 6   1  result-satisfied
;;; 7   1  action-gd
;;; 8   1  syn-item
;;; 9   1  applicable
;;; 10  1  overridden
;;; 11  1  activated
;;; 12  1  succeeded-last
;;; 13  1  marked
;;; 14  1  lcly-cons (short for locally-consistent)
;;; 15  10 lc-consy (rate) (short for local-consistency)

;;; the macros which take data as an argument do NOT work for
;;; setf, the macros which take schemas as arguments do work for setf
;;; note some improvement in performance when using the first form
;;; provided that the data is used for multiple branches/observations

(defmacro schema-data-context-empty              schema-data-context-empty
  (data)
  `(ldb (byte 1 0) ,data))
(defmacro schema-context-empty                        schema--context-empty
  (schema)
  `(ldb (byte 1 0) (schema-data ,schema)))
(defmacro schema-data-context-empty-p          schema-data-context-empty-p
  (data)
  `(flag-truep (schema-data-context-empty ,data)))
(defmacro schema-context-empty-p                    schema-context-empty-p
  (schema)
  `(flag-truep (schema-data-context-empty (schema-data ,schema))))
(defmacro schema-data-context-single            schema-data-context-single
  (data)
  `(ldb (byte 1 1) ,data))
(defmacro schema-context-single                      schema-context-single
  (schema)
  `(ldb (byte 1 1) (schema-data ,schema)))
(defmacro schema-data-context-single-p        schema-data-context-single-p
  (data)
  `(flag-truep (schema-data-context-single ,data)))
(defmacro schema-context-single-p                  schema-context-single-p
  (schema)
  `(flag-truep (schema-data-context-single (schema-data ,schema))))
```

```
(defmacro schema-data-context-conj                  schema-data-context-conj
  (data)
  '(ldb (byte 1 2) ,data))
(defmacro schema-context-conj                            schema-context-conj
  (schema)
  '(ldb (byte 1 2) (schema-data ,schema)))
(defmacro schema-data-context-conj-p              schema-data-context-conj-p
  (data)
  '(flag-truep (schema-data-context-conj ,data)))
(defmacro schema-context-conj-p                        schema-context-conj-p
  (schema)
  '(flag-truep (schema-data-context-conj (schema-data ,schema))))
(defmacro schema-data-result-empty                  schema-data-result-empty
  (data)
  '(ldb (byte 1 3) ,data))
(defmacro schema-result-empty                            schema-result-empty
  (schema)
  '(ldb (byte 1 3) (schema-data ,schema)))
(defmacro schema-data-result-empty-p              schema-data-result-empty-p
  (data)
  '(flag-truep (schema-data-result-empty ,data)))
(defmacro schema-result-empty-p                        schema-result-empty-p
  (schema)
  '(flag-truep (schema-data-result-empty (schema-data ,schema))))
(defmacro schema-data-result-conj                    schema-data-result-conj
  (data)
  '(ldb (byte 1 4) ,data))
(defmacro schema-result-conj                              schema-result-conj
  (schema)
  '(ldb (byte 1 4) (schema-data ,schema)))
(defmacro schema-data-result-conj-p                schema-data-result-conj-p
  (data)
  '(flag-truep (schema-data-result-conj ,data)))
(defmacro schema-result-conj-p                          schema-result-conj-p
  (schema)
  '(flag-truep (schema-data-result-conj (schema-data ,schema))))
(defmacro schema-data-result-negated              schema-data-result-negated
  (data)
  '(ldb (byte 1 5) ,data))
(defmacro schema-result-negated                        schema-result-negated
  (schema)
  '(ldb (byte 1 5) (schema-data ,schema)))
(defmacro schema-data-result-negated-p          schema-data-result-negated-p
  (data)
  '(flag-truep (schema-data-result-negated ,data)))
(defmacro schema-result-negated-p                    schema-result-negated-p
  (schema)
  '(flag-truep (schema-data-result-negated (schema-data ,schema))))
```

```
(defmacro schema-data-result-satisfied          schema-data-result-satisfied
  (data)
  '(ldb (byte 1 6) ,data))
(defmacro schema-result-satisfied                    schema-result-satisfied
  (schema)
  '(ldb (byte 1 6) (schema-data ,schema)))
(defmacro schema-data-result-satisfied-p      schema-data-result-satisfied-p
  (data)
  '(flag-truep (schema-data-result-satisfied ,data)))
(defmacro schema-result-satisfied-p                schema-result-satisfied-p
  (schema)
  '(flag-truep (schema-data-result-satisfied (schema-data ,schema))))
(defmacro schema-data-action-gd                        schema-data-action-gd
  (data)
  '(ldb (byte 1 7) ,data))
(defmacro schema-action-gd                                  schema-action-gd
  (schema)
  '(ldb (byte 1 7) (schema-data ,schema)))
(defmacro schema-data-action-gd-p                    schema-data-action-gd-p
  (data)
  '(flag-truep (schema-data-action-gd ,data)))
(defmacro schema-action-gd-p                              schema-action-gd-p
  (schema)
  '(flag-truep (schema-data-action-gd (schema-data ,schema))))
(defmacro schema-data-syn-item                          schema-data-syn-item
  (data)
  '(ldb (byte 1 8) ,data))
(defmacro schema-syn-item                                    schema-syn-item
  (schema)
  '(ldb (byte 1 8) (schema-data ,schema)))
(defmacro schema-data-syn-item-p                      schema-data-syn-item-p
  (data)
  '(flag-truep (schema-data-syn- item ,data)))
(defmacro schema-syn-item-p                                schema-syn-item-p
  (schema)
  '(flag-truep (schema-data-syn-item (schema-data ,schema))))
(defmacro schema-data-applicable                      schema-data-applicable
  (data)
  '(ldb (byte 1 9) ,data))
(defmacro schema-applicable                                schema-applicable
  (schema)
  '(ldb (byte 1 9) (schema-data ,schema)))
(defmacro schema-data-applicable-p                  schema-data-applicable-p
  (data)
  '(flag-truep (schema-data-applicable ,data)))
(defmacro schema-applicable-p                            schema-applicable-p
  (schema)
  '(flag-truep (schema-data-applicable (schema-data ,schema))))
```

```
(defmacro schema-data-overridden                     schema-data-overridden
  (data)
  `(ldb (byte 1 10) ,data))
(defmacro schema-overridden                          schema-overridden
  (schema)
  `(ldb (byte 1 10) (schema-data ,schema)))
(defmacro schema-data-overridden-p                   schema-data-overridden-p
  (data)
  `(flag-truep (schema-data-overridden ,data)))
(defmacro schema-overridden-p                        schema-overridden-p
  (schema)
  `(flag-truep (schema-data-overridden (schema-data ,schema))))
(defmacro schema-data-activated                      schema-data-activated
  (data)
  `(ldb (byte 1 11) ,data))
(defmacro schema-activated                           schema-activated
  (schema)
  `(ldb (byte 1 11) (schema-data ,schema)))
(defmacro schema-data-activated-p                    schema-data-activated-p
  (data)
  `(flag-truep (schema-data-activated ,data)))
(defmacro schema-activated-p                         schema-activated-p
  (schema)
  `(flag-truep (schema-data-activated (schema-data ,schema))))
(defmacro schema-data-succeeded-last                 schema-data-succeeded-last
  (data)
  `(ldb (byte 1 12) ,data))
(defmacro schema-succeeded-last                      schema-succeeded-last
  (schema)
  `(ldb (byte 1 12) (schema-data ,schema)))
(defmacro schema-data-succeeded-last-p               schema-data-succeeded-last-p
  (data)
  `(flag-truep (schema-data-succeeded-last ,data)))
(defmacro schema-succeeded-last-p                    schema-succeeded-last-p
  (schema)
  `(flag-truep (schema-data-succeeded-last (schema-data ,schema))))
(defmacro schema-data-marked                         schema-data-marked
  (data)
  `(ldb (byte 1 13) ,data))
(defmacro schema-marked                              schema-marked
  (schema)
  `(ldb (byte 1 13) (schema-data ,schema)))
(defmacro schema-data-marked-p                       schema-data-marked-p
  (data)
  `(flag-truep (schema-data-marked ,data)))
(defmacro schema-marked-p                            schema-marked-p
  (schema)
  `(flag-truep (schema-data-marked (schema-data ,schema))))
```

;;; schema datatype functions continued

```
(defmacro schema-data-lcly-cons                     schema-data-lcly-cons
  (data)
  `(ldb (byte 1 14) ,data))
(defmacro schema-lcly-cons                               schema-lcly-cons
  (schema)
  `(ldb (byte 1 14) (schema-data ,schema)))
(defmacro schema-data-lcly-cons-p                 schema-data-lcly-cons-p
  (data)
  `(flag-truep (schema-data-lcly-cons ,data)))
(defmacro schema-lcly-cons-p                           schema-lcly-cons-p
  (schema)
  `(flag-truep (schema-data-lcly-cons (schema-data ,schema))))
(defmacro schema-data-lc-consy                       schema-data-lc-consy
  (data)
  `(ldb (byte 10 15) ,data))
(defmacro schema-lc-consy                                 schema-lc-consy
  (schema)
  `(schema-data-lc-consy (schema-data ,schema)))
(defmacro schema-data-lc-consy-unparse     schema-data-lc-consy-unparse
  (data)
  `(rate-unparse ,data 15))
(defmacro schema-lc-consy-unparse                 schema-lc-consy-unparse
  (schema)
  `(schema-data-lc-consy-unparse (schema-data ,schema)))
(defmacro schema-data-lc-consy-update       schema-data-lc-consy-update
  (data occurred)
  `(rate-update ,data 15 ,occurred))
(defmacro schema-lc-consy-update                   schema-lc-consy-update
  (schema occurred)
  `(schema-data-lc-consy-update (schema-data ,schema) ,occurred))
(defmacro schema-data-lc-consy-high-p       schema-data-lc-consy-high-p
  (data)
  `(rate-high-p ,data 15))
(defmacro schema-lc-consy-high-p                   schema-lc-consy-high-p
  (schema)
  `(schema-data-lc-consy-high-p (schema-data ,schema)))
```

```
;;; schema datatype functions continued
```

schema-array

```
(defvar *schema-array*
  (make-array *schema-maximum*
              :element-type 'schema))

(proclaim '(type (vector schema 3600) *schema-array*))
```

get-schema

```
(defmacro get-schema (index)
  `(svref *schema-array* (the fixnum ,index)))
```

print-schema-enabled

```
(defun print-schema-enabled (schema)
  (schema-format "~A ~15,'0B con ~A"
         (schema-print-name schema)
         (mod (schema-data schema) 32768) ; gets rid of higher order bits
         (schema-lc-consy-unparse schema)))
```

print-schema

```
(defun print-schema (args stream depth)
  (declare (ignore stream depth))
  (if *schema-enabled*
      (print-schema-enabled args)))

;; place result of calling this function in the inital data slot for
;; schema -- NOTE: by default, context and result are marked empty
```

schema-get-initial-data

```
(defun schema-get-initial-data ()
  (let ((foo 0))
    (setf (schema-data-lc-consy foo) (make-rate))
    foo))
```

```
;;; item datatype

;;; used to store information about each primitive and synthetic item
;;; print-name, syn-item-p and code if primitive or
;;; syn-item-index if synthetic define the item and never change
;;; a synthetic item is indicated by having syn-item-p true
;;; a primitive item has code bound to the appropriate microworld code
;;;   (see init-item) which returns the state for the item
;;; generality is the rate of being on rather than off
;;; accessibility is the rate of being reachable by a reliable chain
;;;   of schemas beginning with an applicable schema
;;; gd-created-p indicates if a goal-directed action has been created
;;;   with this item as a goal

;;; note: everything commented out below supports goal-directed action
;;; creation for negated items and is not used currently

(defstruct                                                          item
  (item
   (:constructor make-item-internal
                (print-name code))
   (:print-function print-item))
  (print-name "" :type string)
  (current-state (make-state-unknown) :type fixnum)
  (last-state (make-state-unknown) :type fixnum)
  (code #'always-true :type compiled-function)
  (syn-item-index -1 :type fixnum)
  (data 8 :type fixnum)
  (acc-data 1025 :type fixnum))

;;; data contains important status bits
;;; pos length
;;; 0   1  syn-item
;;; 1   1  gd-pos-created
;;; 2   1  gd-neg-created (currently not used)
;;; 3   10 generality
;;; in acc-data
;;; 0   10 acc-pos
;;; 0   10 acc-neg (currently not used)
;;; note: the stuff not used supports creation of goal-directed actions
;;;   for negated items

;;; the macros which take data as an argument do NOT work for
;;; setf, the macros which take schemas as arguments do work for setf
;;; note some improvement in performance when using the first form
;;; provided that the data is used for multiple branches/observations
;;; item datatype functions

(defmacro item-data-syn-item                          item-data-syn-item
  (data)
  `(ldb (byte 1 0) ,data))
(defmacro item-data-syn-item-p                        item-data-syn-item-p
  (data)
  `(flag-truep (ldb ,(byte 1 0) ,data)))
(defmacro item-data-gd-pos-created                    item-data-gd-pos-created
  (data)
  `(ldb (byte 1 1) ,data))
(defmacro item-data-gd-pos-created-p                  item-data-gd-pos-created-p
  (data)
  `(flag-truep (ldb (byte 1 1) ,data)))
;(defmacro item-data-gd-neg-created                   item-data-gd-neg-created
; (data)
; `(ldb (byte 1 2) ,data))
;(defmacro item-data-gd-neg-created-p                 item-data-gd-neg-created-p
; (data)
; `(flag-truep (ldb (byte 1 2) ,data)))
(defmacro item-data-generality                        item-data-generality
  (data)
  `(ldb (byte 10 3) ,data))
(defmacro item-data-generality-unparse                item-data-generality-unparse
  (data)
  `(rate-unparse ,data 3))
(defmacro item-data-generality-update                 item-data-generality-update
  (data occurred)
  `(rate-update ,data 3 ,occurred))
(defmacro item-data-generality-<                      item-data-generality-<
  (data1 data2)
  `(rate< ,data1 3 ,data2 3))
```

;;; item datatype functions continued

item-acc-data-pos

```
(defmacro item-acc-data-pos
  (data)
  '(ldb (byte 10 0) ,data))
```

item-acc-data-pos-unparse

```
(defmacro item-acc-data-pos-unparse
  (data)
  '(rate-unparse ,data 0))
```

item-acc-data-pos-update

```
(defmacro item-acc-data-pos-update
  (data occurred)
  '(rate-update ,data 0 ,occurred))
```

item-acc-data-pos-high-p

```
(defmacro item-acc-data-pos-high-p
  (data)
  '(rate-high-p ,data 0))
```

item-acc-data-neg

```
;(defmacro item-acc-data-neg
;  (data)
;  '(ldb (byte 10 10) ,data))
```

item-acc-data-neg-unparse

```
;(defmacro item-acc-data-neg-unparse
;  (data)
;  '(rate-unparse ,data 10))
```

item-acc-data-neg-update

```
;(defmacro item-acc-data-neg-update
;  (data occurred)
;  '(rate-update ,data 10 ,occurred))
```

item-acc-data-neg-high-p

```
;(defmacro item-acc-data-neg-high-p
;  (data)
;  '(rate-high-p ,data 10))
```

```
;;; item datatype functions continued

(defmacro item-syn-item (item)                                  item-syn-item
  '(the fixnum (item-data-syn-item (item-data ,itom))))
(defmacro item-syn-item-p (item)                                item-syn-item-p
  '(item-data-syn-item-p (item-data ,item)))
(defmacro item-gd-pos-created (item)                            item-gd-pos-created
  '(the fixnum (item-data-gd-pos-created (item-data ,item))))
(defmacro item-gd-pos-created-p (item)                          item-gd-pos-created-p
  '(item-data-gd-pos-created-p (item-data ,item)))
;(defmacro item-gd-neg-created (item)                           item-gd-neg-created
;  '(the fixnum (item-data-gd-neg-created (item-data ,item))))
;(defmacro item-gd-neg-created-p (item)                         item-gd-neg-created-p
;  '(item-data-gd-neg-created-p (item-data ,item)))
(defmacro item-generality-unparse (item)                        item-generality-unparse
  '(item-data-generality-unparse (item-data ,item)))
(defmacro item-generality-update (item occurred)                item-generality-update
  '(item-data-generality-update (item-data ,item) ,occurred))
(defmacro item-generality-< (item1 item2)                       item-generality-<
  '(item-data-generality-< (item-data ,item1) (item-data ,item2)))
(defmacro item-acc-pos-unparse (item)                           item-acc-pos-unparse
  '(item-acc-data-pos-unparse
    (item-acc-data ,item)))
(defmacro item-acc-pos-update (item occurred)                   item-acc-pos-update
  '(item-acc-data-pos-update
    (item-acc-data ,item) ,occurred))
(defmacro item-acc-pos-high-p (item)                            item-acc-pos-high-p
  '(item-acc-data-pos-high-p
    (item-acc-data, item)))
;(defmacro item-acc-neg-unparse (item)                          item-acc-neg-unparse
;  '(item-acc-data-neg-unparse
;    (item-acc-data ,item)))
;(defmacro item-acc-neg-update (item occurred)                  item-acc-neg-update
;  '(item-acc-data-neg-update
;    (item-acc-data ,item) ,occurred))
;(defmacro item-acc-neg-high-p (item)                           item-acc-neg-high-p
;  '(item-acc-data-neg-high-p
;    (item-acc-data, item)))
```

```
;;; item datatype functions continued

;; run these routines once whenever the data structure changes (either
;; the data structure for item, or the defn of rate), and place results
;; in the initial slots for data above
```

item-data-get-initial

```
(defun item-data-get-initial ()
  (let ((foo 0))
    (setf (item-data-generality foo) (make-rate))
    foo))
```

item-acc-data-get-initial

```
(defun item-acc-data-get-initial ()
  (let ((foo 0))
    (setf (item-acc-data-pos foo) (make-rate))
;   (setf (item-acc-data-neg foo) (make-rate))
    foo))
```

print-item-enabled

```
(defmacro print-item-enabled (item)
  `(let ((data (item-data ,item))
         (a-data (item-acc-data ,item)))
     (item-format "~A:  cur ~A lst ~A gen ~A acc pos ~A ~A~A~A"
; swap with above format string
;            ”~A: cur ~A lst ~A gen ~A acc pos ~A neg ~A ~A~A~A”
                  (item-print-name ,item)
                  (state-unparse (item-current-state ,item))
                  (state-unparse (item-last-state ,item))
                  (item-data-generality-unparse data)
                  (item-acc-data-pos-unparse a-data)
;            (item-acc-data-neg-unparse a-data)
                  (if (item-data-syn-item-p data)
                      "syn "
                    "")
                  (if (item-data-gd-pos-created-p data)
                      "ca+ "
                    "")
;            (if (item-data-gd-neg-created-p data)
;                ”gd- ”
;              ””)
                  )))
```

```
;;; item datatype functions continued
```

item-array

```
(defvar *item-array*
  (make-array *item-maximum*
              :element-type 'item))

(proclaim '(type (vector item 500) *item-array*))

;; note: both forms are given for efficiency
;; (when manipulating an item several times, better to get item
;; directly using get-item rather than repeatedly using the longer form)
```

get-item

```
(defmacro get-item (x)
  `(svref *item-array* (the fixnum ,x)))
```

get-item-current-state

```
(defmacro get-item-current-state (x)
  `(the fixnum (item-current-state (get-item ,x))))
```

get-item-last-state

```
(defmacro get-item-last-state (x)
  `(the fixnum (item-last-state (get-item ,x))))
```

get-item-code

```
(defmacro get-item-code (x)
  `(item-code (get-item ,x)))
```

make-item

```
(defun make-item (print-name code)
  (if (fix= *item-number* *item-maximum*)
      (error "no more space for primitive or synthetic items in item array")
    (progn
      (setf (get-item *item-number*)
            (make-item-internal print-name code)
            *item-number*
            (fix1+ *item-number*))
      (fix1- *item-number*))))
```

```
;;; conj datatype

;;; used to keep the state of conjunctions of items (primitive or
;;; synthetic)
;;; positive/negative-flag-arrays have a flag set if that item is
;;;   included positively/negatively in the item-array
;;; inclusion-array indicates which other conjs are included by a
;;;   particular conj (i.e. a&b&c includes a&b and a&c, etc.)
```

conj

```
(defstruct
  (conj
   (:constructor make-conj-internal)
   (:print-function print-conj)
   )
  (print-name "" :type string)
  (current-state (make-state-unknown) :type fixnum)
  (last-state (make-state-unknown) :type fixnum)
  (item-array (make-state-array 25) :type (vector fixnum 25))
  (pos-flag-array (make-flag-array 10) :type (vector fixnum 10))
  (neg-flag-array (make-flag-array 10) :type (vector fixnum 10))
  (inclusion-array (make-flag-array 10) :type (vector fixnum 10))
  (data 4 :type fixnum))
```

```
;;; conj datatype functions

;;; data packing similar to item datatype
;;; pos length
;;; 0  1  gd-pos-created
;;; bit 1 is not used
;;; 2  10 accessibility-pos

;;; the macros which take data as an argument do NOT work for
;;; setf, the macros which take schemas as arguments do work for setf
;;; note some improvement in performance when using the first form
;;; provided that the data is used for multiple branches/observations

(defmacro conj-data-gd-pos-created (data)          conj-data-gd-pos-created
  `(ldb (byte 1 0) ,data))
(defmacro conj-data-gd-pos-created-p (data)        conj-data-gd-pos-created-p
  `(flag-truep (ldb (byte 1 0) ,data)))

(defmacro conj-data-acc-pos                        conj-data-acc-pos
  (data)
  `(ldb (byte 10 2) ,data))
(defmacro conj-data-acc-pos-unparse                conj-data-acc-pos-unparse
  (data)
  `(rate-unparse ,data 2))
(defmacro conj-data-acc-pos-update                 conj-data-acc-pos-update
  (data occurred)
  `(rate-update ,data 2 ,occurred))
(defmacro conj-data-acc-pos-high-p                 conj-data-acc-pos-high-p
  (data)
  `(rate-high-p ,data 2))

(defmacro conj-gd-pos-created (conj)               conj-gd-pos-created
  `(conj-data-gd-pos-created (conj-data ,conj)))
(defmacro conj-gd-pos-created-p (conj)             conj-gd-pos-created-p
  `(conj-data-gd-pos-created-p (conj-data ,conj)))

(defmacro conj-acc-pos-unparse (conj)              conj-acc-pos-unparse
  `(conj-data-acc-pos-unparse
    (conj-data ,conj)))
(defmacro conj-acc-pos--update (conj occurred)     conj-acc-pos-update
  `(conj-data-acc-pos-update
    (conj-data ,conj) ,occurred))
(defmacro conj-acc-pos-high-p (conj)               conj-acc-pos-high-p
  `(conj-data-acc-pos-high-p
    (conj-data ,conj)))
```

```
;;; conj datatype functions continued

;; run this routine once whenever the data structure changes (either
;; the data structure for conj, or the defn of rate), and place
;; result in the initial slot for data above
```

conj-data-get-initial

```
(defun conj-data-get-initial ()
  (let ((foo 0))
    (setf (conj-data-acc-pos foo) (make-rate))
    foo))
```

conj-inclusion-array-unparse

```
(defmacro conj-inclusion-array-unparse
  (array)

  `(let ((result ""))
     (dotimes
      (x *conj-number* (string-right-trim "&" result))
      (let ((flag (flag-array-get-flag ,array x)))
        (if (flag-truep flag)
            (setq result
                  (concatenate 'string
                               result
                               (format nil "~D" x)
                               "&")))))))
```

print-conj-enabled

```
(defmacro print-conj-enabled (conj)
  `(let ((data (conj-data ,conj)))
     (conj-format "~A:  cur ~A lst ~A acc ~A  ~4A incl ~A"
                  (conj-print-name ,conj)
                  (state-unparse (conj-current-state ,conj))
                  (state-unparse (conj-last-state ,conj))
                  (conj-data-acc-pos-unparse data)
                  (if (conj-data-gd-pos-created-p data)
                      "ca+"
                    "")
                  (conj-inclusion-array-unparse
                   (conj-inclusion-array ,conj)))))
```

print-conj

```
(defun print-conj (args stream depth)
  (declare (ignore stream depth))
  (if *conj-enabled*
      (print-conj-enabled args)))
```

```
;;; conj datatype functions continued
```

conj-array

```
(defvar *conj-array*
  (make-array *conj-maximum*
              :element-type 'conj))

(proclaim '(type (vector conj 240) *conj-array*))

;; note: both forms are given for efficiency (when manipulating an
;; conjunction several times, better to get conjunction directly using
;; get-conj rather than repeatedly using the longer form)
```

get-conj

```
(defmacro get-conj (x)
  `(svref *conj-array* (the fixnum ,x)))
```

get-conj-current-state

```
(defmacro get-conj-current-state (x)
  `(the fixnum (conj-current-state (get-conj ,x))))
```

get-conj-last-state

```
(defmacro get-conj-last-state (x)
  `(the fixnum (conj-last-state (get-conj ,x))))
```

conj-update-print-name

```
(defmacro conj-update-print-name (conj)
  `(setf (conj-print-name ,conj)
         (concatenate 'string
                      "("
                      (state-array-get-print-name
                       (conj-item-array ,conj))
                      ")")))
```

conj-find

```
(defmacro conj-find (item-array)
  `(dotimes
    (x *conj-number* nil)
    (let ((compare-array (conj-item-array (get-conj x))))
      (if (dotimes
           (y 25 t)
           (if (fix/= (aref compare-array y) (aref ,item-array y))
               (return nil)))
          (return x)))))
```

```
;;; conj datatype functions continued

;; inclusion array simply has an ON flag for every included
;; conjunction
;; need to both update all old conjunctions with marks for a new one,
;; and update the new one with marks for all
;; NOTE: the inclusion array is *only* for conjunctions --- items can
;; be checked directly by looking at the item array
;; NOTE: this routine expects to be called as the new conjunction has
;; been added to the conjunction array, but before
;; *conj-number* has been incremented
```

conj-update-inclusion-array

```
(defmacro conj-update-inclusion-array (conj)
  `(let ((item-array (conj-item-array ,conj))
         (inclusion-array (conj-inclusion-array ,conj))
         (pos *conj-number*))
     (dotimes
      (x *conj-number*)
      (let* ((current-conj (get-conj x))
             (current-array (conj-item-array current-conj))
             (current-inclusion-array
              (conj-inclusion-array current-conj)))
        ;; update info for old schemas
        (if (state-array-included-p current-array item-array 20)
            (setf (flag-array-get-flag current-inclusion-array pos)
                  (make-flag-true)))
        ;; update info for newer schema
        (if (state-array-included-p item-array current-array 20)
            (setf (flag-array-get-flag inclusion-array x)
                  (make-flag-true)))))
     ;; set own bit ON as it includes itself
     (setf (flag-array-get-flag inclusion-array pos)
           (make-flag-true))))
```

```
;;; conj datatype functions continued

;;; make-conj returns number of matching existing conjunction,
;;; or creates and returns the number of a new conjunction if needed
```

make-conj

```
(defun make-conj (item-array)
  (if (fix= *conj-number* *conj-maximum*)
      (error "no more space for conjs in conj array")
    (let ((check (conj-find item-array)))
      (if check
          check
        (let ((current-conj
                (setf (get-conj *conj-number*)
                      (make-conj-internal))))
          (state-array-copy
           item-array
           (conj-item-array current-conj) 20)
          (state-array-copy-pos-flag
           item-array
           (conj-pos-flag-array current-conj) 20)
          (state-array-copy-neg-flag
           item-array
           (conj-neg-flag-array current-conj) 20)
          (conj-update-print-name current-conj)
          (conj-update-inclusion-array current-conj)
          (main-format "~5D conj-created ~D ~A~%"
                       *clock-tick* *conj-number*
                       (conj-print-name current-conj))
          (setf *conj-number* (fix1+ *conj-number*))
          (fix1- *conj-number*))))))
```

```
;;; syn-item datatype

;;; synthetic items are used to designate validity conditions of
;;; unreliable schemas which are locally consistent
;;; the unreliable schema which causes creation is called the
;;; host-schema, while the synthetic item is the schemas "reifier"
;;; synthetic items are included as items in the item array and are
;;; treated 100% as if they were primitive items (can be in
;;; context/result and conjunctions too) however the state of a
;;; synthetic item is determined by the schema mechanism rather than
;;; by a call to a microworld function
;;; host-schema is the index into the schema array for the host schema
;;; item-index is the index into the item array for entry for this
;;;   synthetic item
;;; maybe-state is used by the routines which calculate the state for
;;;   the synthetic items -- it is merely a place to stash an
;;;   intermediate value
;;; on-duration and off-duration is the length of time the synthetic
;;;   item tends to stay on or off once placed in that state
;;; set-time is the clock tick when the item was last modified
;;; unknown-time is the clock tick when the item should be
;;;   automatically set unknown

(defstruct                                                        syn-item
  (syn-item
   (:constructor make-syn-item-internal
                 (host-schema)))
  (host-schema -1 :type fixnum)
  (item-index -1 :type fixnum)
  (current-state (make-state-unknown) :type fixnum)
  (maybe-state (make-state-unknown) :type fixnum)
  (on-duration (make-average) :type average)
  (off-duration (make-average) :type average)
  (set-time -1 :type fixnum)
  (unknown-time -1 :type fixnum))
```

```
;;; syn-item datatype functions
```

syn-item-array

```
(defvar *syn-item-array*
  (make-array *syn-item-maximum*
              :element-type 'syn-item))

(proclaim '(type (vector syn-item 350) *syn-item-array*))

;; note: both forms are given for efficiency (when manipulating
;; several times, better to get syn-item directly rather then
;; repeatedly using the longer form)
```

get-syn-item

```
(defmacro get-syn-item (x)
  `(svref *syn-item-array* (the fixnum ,x)))
```

get-syn-item-current-state

```
(defmacro get-syn-item-current-state (x)
  `(the fixnum (syn-item-current-state (get-syn-item ,x))))
```

print-syn-item-enabled

```
(defmacro print-syn-item-enabled (item)
  `(let ((data (item-data ,item))
         (syn-item (get-syn-item (item-syn-item-index ,item))))
     (syn-item-format
      "~A host ~4D item ~4D:  cur ~A lst ~A int ~A gen ~A~%~
       ~10Tdur ~4D set ~4D off ~4D unk ~4D"
      (item-print-name ,item)
      (syn-item-host-schema syn-item)
      (syn-item-item-index syn-item)
      (state-unparse (item-data-current-state data))
      (state-unparse (item-data-last-state data))
      (state-unparse (syn-item-current-state syn-item))
      (item-data-generality-unparse data)
      (syn-item-duration syn-item)
      (syn-item-set-time syn-item)
      (syn-item-off-time syn-item)
      (syn-item-unknown-time syn-item))))
```

print-item

```
(defun print-item (args stream depth)
  (declare (ignore stream depth))
  (if (and *syn-item-enabled*
           (item-syn-item-p args))
      (print-syn-item-enabled args)
    (if *item-enabled*
        (print-item-enabled args))))
```

make–syn–item

```
(defun make-syn-item (host-schema)
  (declare (fixnum host-schema))
  (if (fix= *syn-item-number* *syn-item-maximum*)
      (error "no more space for synthetic items in syn-item array")
    (let* ((schema (get-schema host-schema))
           (syn-item (make-syn-item-internal host-schema))
           (item-index (make-item
                        (concatenate 'string
                                     "["
                                     (schema-print-name schema)
                                     "]")
                        (compile nil
                                 `(lambda ()
                                    ,`(make-state-both)))))
           (current-item (get-item item-index)))
      (setf (item-syn-item current-item)
            (make-flag-true)
            (item-syn-item-index current-item)
            *syn-item-number*
            (syn-item-item-index syn-item)
            item-index
            (get-syn-item *syn-item-number*)
            syn-item)
      (main-format "~5D syn item created ~D ~A~%"
                   *clock-tick*
                   *syn-item-number*
                   (item-print-name current-item))
      (setf *syn-item-number* (fix1+ *syn-item-number*))))
  (fix1- *syn-item-number*))
```

```
;;; action datatype

;;; a simple way to keep the various microworld actions and the
;;; corresponding human-readable print name

(defvar *action-array*                                   *action-array*
  (make-array 25 :element-type 'compiled-function))
(defvar *action-print-name-array*             *action-print-name-array*
  (make-array 25 :element-type 'string))

(proclaim '(type (vector compiled-function 25) *action-array*))
(proclaim '(type (vector string 25) *action-print-name-array*))

(defmacro get-action (index)                                 get-action
  `(aref *action-array* ,index))

(defmacro get-action-print-name (index)           get-action-print-name
  `(aref *action-print-name-array* ,index))

(defun make-action (print-name compiled-code)               make-action
  (if (fix= *action-number* *action-maximum*)
      (error "no more space for actions in action array")
    (progn
      (setf (get-action *action-number*)
            compiled-code
            (get-action-print-name *action-number*)
            print-name
            *action-number*
            (fix1+ *action-number*))
      (fix1- *action-number*))))
```

schema-update-print-name

```
(defun schema-update-print-name (schema)
  (let ((data (schema-data schema))
        (result-item (schema-result-item schema))
        (action-item (schema-action-item schema))
        (context-item (schema-context-item schema)))
    (setf (schema-print-name schema)
        (concatenate 'string
                     (if (not (schema-context-empty-p schema))
                         (if (schema-data-context-conj-p data)
                             (conj-print-name
                              (get-conj context-item))
                           (state-array-get-print-name
                            (schema-context-array schema))))
                     (if (schema-data-action-gd-p data)
                         (goal-directed-action-print-name
                          (get-gd-action action-item))
                       (get-action-print-name action-item))
                     (if (and (not (schema-result-empty-p schema))
                              (not (schema-result-conj-p schema))
                              (schema-result-negated-p schema))
                         "-")
                     (if (not (schema-result-empty-p schema))
                         (if (schema-data-result-conj-p data)
                             (conj-print-name
                              (get-conj result-item))
                           (item-print-name (get-item result-item))))))))
```

make-action-schema

```
(defun make-action-schema
  (action-index)

  (if (= *schema-number* *schema-maximum*)
      (error "no more space for schemas")
    (prog1
        (setf (get-schema *schema-number*)
              (make-schema-internal
               action-index
               (get-action action-index)))
      (setf *schema-number* (fix1+ *schema-number*)))))
```

make-blank-action-schema

```
(defun make-blank-action-schema
  (action-index)

  (let ((new-schema (make-action-schema action-index)))
    (setf (schema-result-empty new-schema) (make-flag-true)
          (schema-context-empty new-schema) (make-flag-true))
    (schema-update-print-name new-schema)
    new-schema))
```

;;; major function: init–everything

init–everything

```
(defun init-everything (randomstatefilename selector)
  (setf *clock-tick*
        (the fixnum
             (cond
              ((fix= 0 selector)
               (microworld:init-world randomstatefilename))
              ((fix= 1 selector)
               (microworld:init-world-no-right-object randomstatefilename))
              ((fix= 2 selector)
               (microworld:init-world-no-left-object randomstatefilename))
              (t (error "schema:init-everything -- incorrect selector")))))
  (setf *conj-number* 0)
  (init-item)
  (init-schema-and-action)
  'init-everything-finished)
```

init-item

```
;;; initialization function: init-item

(defun init-item ()
  (setf *item-number* 0)
  (setf *syn-item-number* 0)

  ;; order matters for the show-item debugging routine
  ;; needs to be in standard x,y format for display

  (make-item "hp00" #'microworld:hp00)  ;  0
  (make-item "hp10" #'microworld:hp10)  ;  1
  (make-item "hp20" #'microworld:hp20)  ;  2
  (make-item "hp01" #'microworld:hp01)  ;  3
  (make-item "hp11" #'microworld:hp11)  ;  4
  (make-item "hp21" #'microworld:hp21)  ;  5
  (make-item "hp02" #'microworld:hp02)  ;  6
  (make-item "hp12" #'microworld:hp12)  ;  7
  (make-item "hp22" #'microworld:hp22)  ;  8

  (make-item "vp00" #'microworld:vp00)  ;  9
  (make-item "vp10" #'microworld:vp10)  ; 10
  (make-item "vp20" #'microworld:vp20)  ; 11
  (make-item "vp01" #'microworld:vp01)  ; 12
  (make-item "vp11" #'microworld:vp11)  ; 13
  (make-item "vp21" #'microworld:vp21)  ; 14
  (make-item "vp02" #'microworld:vp02)  ; 15
  (make-item "vp12" #'microworld:vp12)  ; 16
  (make-item "vp22" #'microworld:vp22)  ; 17

  (make-item "vf00" #'microworld:vf00)  ; 18
  (make-item "vf10" #'microworld:vf10)  ; 19
  (make-item "vf20" #'microworld:vf20)  ; 20
  (make-item "vf30" #'microworld:vf30)  ; 21
  (make-item "vf40" #'microworld:vf40)  ; 22
  (make-item "vf01" #'microworld:vf01)  ; 23
  (make-item "vf11" #'microworld:vf11)  ; 24
  (make-item "vf21" #'microworld:vf21)  ; 25
  (make-item "vf31" #'microworld:vf31)  ; 26
  (make-item "vf41" #'microworld:vf41)  ; 27
  (make-item "vf02" #'microworld:vf02)  ; 28
  (make-item "vf12" #'microworld:vf12)  ; 29
  (make-item "vf22" #'microworld:vf22)  ; 30
  (make-item "vf32" #'microworld:vf32)  ; 31
  (make-item "vf42" #'microworld:vf42)  ; 32
  (make-item "vf03" #'microworld:vf03)  ; 33
  (make-item "vf13" #'microworld:vf13)  ; 34
  (make-item "vf23" #'microworld:vf23)  ; 35
  (make-item "vf33" #'microworld:vf33)  ; 36
  (make-item "vf43" #'microworld:vf43)  ; 37
  (make-item "vf04" #'microworld:vf04)  ; 38
  (make-item "vf14" #'microworld:vf14)  ; 39
  (make-item "vf24" #'microworld:vf24)  ; 40
  (make-item "vf34" #'microworld:vf34)  ; 41
  (make-item "vf44" #'microworld:vf44)  ; 42
```

```
;;; init-item continued

    ;; defined in the same order as in the microworld
    ;; so show-item gives the same visual appearance
    ;; as the object definition in the file

    (make-item "fovf00" #'microworld:fovf00)  ; 43
    (make-item "fovf01" #'microworld:fovf01)  ; 44
    (make-item "fovf02" #'microworld:fovf02)  ; 45
    (make-item "fovf03" #'microworld:fovf03)  ; 46
    (make-item "fovf10" #'microworld:fovf10)  ; 47
    (make-item "fovf11" #'microworld:fovf11)  ; 48
    (make-item "fovf12" #'microworld:fovf12)  ; 49
    (make-item "fovf13" #'microworld:fovf13)  ; 50
    (make-item "fovf20" #'microworld:fovf20)  ; 51
    (make-item "fovf21" #'microworld:fovf21)  ; 52
    (make-item "fovf22" #'microworld:fovf22)  ; 53
    (make-item "fovf23" #'microworld:fovf23)  ; 54
    (make-item "fovf30" #'microworld:fovf30)  ; 55
    (make-item "fovf31" #'microworld:fovf31)  ; 56
    (make-item "fovf32" #'microworld:fovf32)  ; 57
    (make-item "fovf33" #'microworld:fovf33)  ; 58

    (make-item "fovb00" #'microworld:fovb00)  ; 59
    (make-item "fovb01" #'microworld:fovb01)  ; 60
    (make-item "fovb02" #'microworld:fovb02)  ; 61
    (make-item "fovb03" #'microworld:fovb03)  ; 62
    (make-item "fovb10" #'microworld:fovb10)  ; 63
    (make-item "fovb11" #'microworld:fovb11)  ; 64
    (make-item "fovb12" #'microworld:fovb12)  ; 65
    (make-item "fovb13" #'microworld:fovb13)  ; 66
    (make-item "fovb20" #'microworld:fovb20)  ; 67
    (make-item "fovb21" #'microworld:fovb21)  ; 68
    (make-item "fovb22" #'microworld:fovb22)  ; 69
    (make-item "fovb23" #'microworld:fovb23)  ; 70
    (make-item "fovb30" #'microworld:fovb30)  ; 71
    (make-item "fovb31" #'microworld:fovb31)  ; 72
    (make-item "fovb32" #'microworld:fovb32)  ; 73
    (make-item "fovb33" #'microworld:fovb33)  ; 74
```

;;; init–item continued

```
(make-item "fovl00" #'microworld:fovl00)  ;  75
(make-item "fovl01" #'microworld:fovl01)  ;  76
(make-item "fovl02" #'microworld:fovl02)  ;  77
(make-item "fovl03" #'microworld:fovl03)  ;  78
(make-item "fovl10" #'microworld:fovl10)  ;  79
(make-item "fovl11" #'microworld:fovl11)  ;  80
(make-item "fovl12" #'microworld:fovl12)  ;  81
(make-item "fovl13" #'microworld:fovl13)  ;  82
(make-item "fovl20" #'microworld:fovl20)  ;  83
(make-item "fovl21" #'microworld:fovl21)  ;  84
(make-item "fovl22" #'microworld:fovl22)  ;  85
(make-item "fovl23" #'microworld:fovl23)  ;  86
(make-item "fovl30" #'microworld:fovl30)  ;  87
(make-item "fovl31" #'microworld:fovl31)  ;  88
(make-item "fovl32" #'microworld:fovl32)  ;  89
(make-item "fovl33" #'microworld:fovl33)  ;  90

(make-item "fovr00" #'microworld:fovr00)  ;  91
(make-item "fovr01" #'microworld:fovr01)  ;  92
(make-item "fovr02" #'microworld:fovr02)  ;  93
(make-item "fovr03" #'microworld:fovr03)  ;  94
(make-item "fovr10" #'microworld:fovr10)  ;  95
(make-item "fovr11" #'microworld:fovr11)  ;  96
(make-item "fovr12" #'microworld:fovr12)  ;  97
(make-item "fovr13" #'microworld:fovr13)  ;  98
(make-item "fovr20" #'microworld:fovr20)  ;  99
(make-item "fovr21" #'microworld:fovr21)  ; 100
(make-item "fovr22" #'microworld:fovr22)  ; 101
(make-item "fovr23" #'microworld:fovr23)  ; 102
(make-item "fovr30" #'microworld:fovr30)  ; 103
(make-item "fovr31" #'microworld:fovr31)  ; 104
(make-item "fovr32" #'microworld:fovr32)  ; 105
(make-item "fovr33" #'microworld:fovr33)  ; 106

(make-item "fovx00" #'microworld:fovx00)  ; 107
(make-item "fovx01" #'microworld:fovx01)  ; 108
(make-item "fovx02" #'microworld:fovx02)  ; 109
(make-item "fovx03" #'microworld:fovx03)  ; 110
(make-item "fovx10" #'microworld:fovx10)  ; 111
(make-item "fovx11" #'microworld:fovx11)  ; 112
(make-item "fovx12" #'microworld:fovx12)  ; 113
(make-item "fovx13" #'microworld:fovx13)  ; 114
(make-item "fovx20" #'microworld:fovx20)  ; 115
(make-item "fovx21" #'microworld:fovx21)  ; 116
(make-item "fovx22" #'microworld:fovx22)  ; 117
(make-item "fovx23" #'microworld:fovx23)  ; 118
(make-item "fovx30" #'microworld:fovx30)  ; 119
(make-item "fovx31" #'microworld:fovx31)  ; 120
(make-item "fovx32" #'microworld:fovx32)  ; 121
(make-item "fovx33" #'microworld:fovx33)  ; 122
```

```
;;; init-item continued

  (make-item "tactf" #'microworld:tactf)  ; 123
  (make-item "tactb" #'microworld:tactb)  ; 124
  (make-item "tactr" #'microworld:tactr)  ; 125
  (make-item "tactl" #'microworld:tactl)  ; 126

  (make-item "bodyf" #'microworld:bodyf)  ; 127
  (make-item "bodyb" #'microworld:bodyb)  ; 128
  (make-item "bodyr" #'microworld:bodyr)  ; 129
  (make-item "bodyl" #'microworld:bodyl)  ; 130

  (make-item "text0" #'microworld:text0)  ; 131
  (make-item "text1" #'microworld:text1)  ; 132
  (make-item "text2" #'microworld:text2)  ; 133
  (make-item "text3" #'microworld:text3)  ; 134

  (make-item "taste0" #'microworld:taste0)  ; 135
  (make-item "taste1" #'microworld:taste1)  ; 136
  (make-item "taste2" #'microworld:taste2)  ; 137
  (make-item "taste3" #'microworld:taste3)  ; 138

  (make-item "hcl" #'microworld:hcl)  ; 139
  (make-item "hgr" #'microworld:hgr)  ; 140
  )

;;; initialization function: init-schema-and-action

(defun init-schema-and-action ()                    init-schema-and-action
  (setf *action-number* 0)
  (setf *schema-number* 0)
  (make-blank-action-schema (make-action "/handf/" #'microworld:handf))
  (make-blank-action-schema (make-action "/handb/" #'microworld:handb))
  (make-blank-action-schema (make-action "/handr/" #'microworld:handr))
  (make-blank-action-schema (make-action "/handl/" #'microworld:handl))
  (make-blank-action-schema (make-action "/eyef/" #'microworld:eyef))
  (make-blank-action-schema (make-action "/eyeb/" #'microworld:eyeb))
  (make-blank-action-schema (make-action "/eyer/" #'microworld:eyer))
  (make-blank-action-schema (make-action "/eyel/" #'microworld:eyel))
  (make-blank-action-schema (make-action "/grasp/" #'microworld:grasp))
  (make-blank-action-schema (make-action "/ungrasp/" #'microworld:ungrasp)))

;;; output function: show-items

(defmacro show-items ()                                         show-items
  (if *show-items-enabled*
      '(show-items-enabled)))
```

;;; output function: show–items–enabled

show–items–enabled

```
(defun show-items-enabled ()
  (format *output-stream*
        "hp     vp     vf        fovf  fovb  fovl  fovr  fovx    ~
         tact  body~%")
  (format *output-stream*
        "~A~A~A    ~A~A~A    ~A~A~A~A~A    ~A~A~A~A  ~
         ~A~A~A~A  ~A~A~A~A  ~A~A~A~A  ~A~A~A~A      ~
         ~A     ~A~%"
        (state-unparse (get-item-current-state 6))
        (state-unparse (get-item-current-state 7))
        (state-unparse (get-item-current-state 8))

        (state-unparse (get-item-current-state 15))
        (state-unparse (get-item-current-state 16))
        (state-unparse (get-item-current-state 17))

        (state-unparse (get-item-current-state 38))
        (state-unparse (get-item-current-state 39))
        (state-unparse (get-item-current-state 40))
        (state-unparse (get-item-current-state 41))
        (state-unparse (get-item-current-state 42))

        (state-unparse (get-item-current-state 55))
        (state-unparse (get-item-current-state 56))
        (state-unparse (get-item-current-state 57))
        (state-unparse (get-item-current-state 58))

        (state-unparse (get-item-current-state 71))
        (state-unparse (get-item-current-state 72))
        (state-unparse (get-item-current-state 73))
        (state-unparse (get-item-current-state 74))

        (state-unparse (get-item-current-state 87))
        (state-unparse (get-item-current-state 88))
        (state-unparse (get-item-current-state 89))
        (state-unparse (get-item-current-state 90))

        (state-unparse (get-item-current-state 103))
        (state-unparse (get-item-current-state 104))
        (state-unparse (get-item-current-state 105))
        (state-unparse (get-item-current-state 106))

        (state-unparse (get-item-current-state 119))
        (state-unparse (get-item-current-state 120))
        (state-unparse (get-item-current-state 121))
        (state-unparse (get-item-current-state 122))

        (state-unparse (get-item-current-state 123))

        (state-unparse (get-item-current-state 127)))
```

```
;;; show-items-enabled continued

  (format *output-stream*
          "~A~A~A    ~A~A~A    ~A~A~A~A~A    ~A~A~A~A ~
           ~A~A~A~A ~A~A~A~A ~A~A~A~A ~A~A~A~A    ~
           ~A ~A   ~A ~A~%"
          (state-unparse (get-item-current-state 3))
          (state-unparse (get-item-current-state 4))
          (state-unparse (get-item-current-state 5))

          (state-unparse (get-item-current-state 12))
          (state-unparse (get-item-current-state 13))
          (state-unparse (get-item-current-state 14))

          (state-unparse (get-item-current-state 33))
          (state-unparse (get-item-current-state 34))
          (state-unparse (get-item-current-state 35))
          (state-unparse (get-item-current-state 36))
          (state-unparse (get-item-current-state 37))

          (state-unparse (get-item-current-state 51))
          (state-unparse (get-item-current-state 52))
          (state-unparse (get-item-current-state 53))
          (state-unparse (get-item-current-state 54))

          (state-unparse (get-item-current-state 67))
          (state-unparse (get-item-current-state 68))
          (state-unparse (get-item-current-state 69))
          (state-unparse (get-item-current-state 70))

          (state-unparse (get-item-current-state 83))
          (state-unparse (get-item-current-state 84))
          (state-unparse (get-item-current-state 85))
          (state-unparse (get-item-current-state 86))

          (state-unparse (get-item-current-state 99))
          (state-unparse (get-item-current-state 100))
          (state-unparse (get-item-current-state 101))
          (state-unparse (get-item-current-state 102))

          (state-unparse (get-item-current-state 115))
          (state-unparse (get-item-current-state 116))
          (state-unparse (get-item-current-state 117))
          (state-unparse (get-item-current-state 118))

          (state-unparse (get-item-current-state 126))
          (state-unparse (get-item-current-state 125))

          (state-unparse (get-item-current-state 130))
          (state-unparse (get-item-current-state 129)))
```

;;; show-items-enabled continued

```
  (format *output-stream*
          "~A~A~A    ~A~A~A   ~A~A~A~A~A    ~A~A~A~A ~
           ~A~A~A~A  ~A~A~A~A ~A~A~A~A  ~A~A~A~A    ~
           ~A     ~A~%"
          (state-unparse (get-item-current-state 0))
          (state-unparse (get-item-current-state 1))
          (state-unparse (get-item-current-state 2))

          (state-unparse (get-item-current-state 9))
          (state-unparse (get-item-current-state 10))
          (state-unparse (get-item-current-state 11))

          (state-unparse (get-item-current-state 28))
          (state-unparse (get-item-current-state 29))
          (state-unparse (get-item-current-state 30))
          (state-unparse (get-item-current-state 31))
          (state-unparse (get-item-current-state 32))

          (state-unparse (get-item-current-state 47))
          (state-unparse (get-item-current-state 48))
          (state-unparse (get-item-current-state 49))
          (state-unparse (get-item-current-state 50))

          (state-unparse (get-item-current-state 63))
          (state-unparse (get-item-current-state 64))
          (state-unparse (get-item-current-state 65))
          (state-unparse (get-item-current-state 66))

          (state-unparse (get-item-current-state 79))
          (state-unparse (get-item-current-state 80))
          (state-unparse (get-item-current-state 81))
          (state-unparse (get-item-current-state 82))

          (state-unparse (get-item-current-state 95))
          (state-unparse (get-item-current-state 96))
          (state-unparse (get-item-current-state 97))
          (state-unparse (get-item-current-state 98))

          (state-unparse (get-item-current-state 111))
          (state-unparse (get-item-current-state 112))
          (state-unparse (get-item-current-state 113))
          (state-unparse (get-item-current-state 114))

          (state-unparse (get-item-current-state 124))

          (state-unparse (get-item-current-state 128)))
```

;;; show–items–enabled continued

```
 (format *output-stream*
        "                ~A~A~A~A~A    ~A~A~A~A  ~
         ~A~A~A~A  ~A~A~A~A  ~A~A~A~A  ~A~A~A~A    ~
         ~A  ~A   ~A  ~A~%"

        (state-unparse (get-item-current-state 23))
        (state-unparse (get-item-current-state 24))
        (state-unparse (get-item-current-state 25))
        (state-unparse (get-item-current-state 26))
        (state-unparse (get-item-current-state 27))

        (state-unparse (get-item-current-state 43))
        (state-unparse (get-item-current-state 44))
        (state-unparse (get-item-current-state 45))
        (state-unparse (get-item-current-state 46))

        (state-unparse (get-item-current-state 59))
        (state-unparse (get-item-current-state 60))
        (state-unparse (get-item-current-state 61))
        (state-unparse (get-item-current-state 62))

        (state-unparse (get-item-current-state 75))
        (state-unparse (get-item-current-state 76))
        (state-unparse (get-item-current-state 77))
        (state-unparse (get-item-current-state 78))

        (state-unparse (get-item-current-state 91))
        (state-unparse (get-item-current-state 92))
        (state-unparse (get-item-current-state 93))
        (state-unparse (get-item-current-state 94))

        (state-unparse (get-item-current-state 107))
        (state-unparse (get-item-current-state 108))
        (state-unparse (get-item-current-state 109))
        (state-unparse (get-item-current-state 110))
        "text"
        "taste"
        "hcl"
        "hgr")
```

```
;;; show-items-enabled continued

  (format *output-stream*
          "             ~A~A~A~A~A~
          ~A~A~A~A~A  ~A~A~A~A     ~A    ~A~%"

          (state-unparse (get-item-current-state 18))
          (state-unparse (get-item-current-state 19))
          (state-unparse (get-item-current-state 20))
          (state-unparse (get-item-current-state 21))
          (state-unparse (get-item-current-state 22))

          "                                  "

          (state-unparse (get-item-current-state 131))
          (state-unparse (get-item-current-state 132))
          (state-unparse (get-item-current-state 133))
          (state-unparse (get-item-current-state 134))

          (state-unparse (get-item-current-state 135))
          (state-unparse (get-item-current-state 136))
          (state-unparse (get-item-current-state 137))
          (state-unparse (get-item-current-state 138))

          (state-unparse (get-item-current-state 139))

          (state-unparse (get-item-current-state 140)))

  (format *output-stream* "~%"))
```

;;; major function: schema-update-applicable

```
(defmacro context-satisfied-p (array)                          context-satisfied-p
  `(state-array-included-p *microworld-state* ,array 25))
(defmacro conj-satisfied-p (conj-index)                           conj-satisfied-p
  `(state-on-p (conj-current-state
               (get-conj ,conj-index))))
```

;;; for each schema with satisfied context, mark it as applicable
;;; an empty context is always satisfied, otherwise, it is satisfied if
;;; a) the context has been made into a conj and the conj is ON
;;; b) each state in the context is matched by the microworld state

```
(defun schema-update-applicable ()                        schema-update-applicable
  (applicable-format "updating applicable~%")
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (setf (schema-applicable schema)
           (flag-parse
            (or (schema-context-empty-p schema)
                (if (schema-context-conj-p schema)
                    (conj-satisfied-p
                     (schema-context-item schema))
                  (context-satisfied-p (schema-context-array schema))))))
     (applicable-format "~4D ~25A~A"
                        schema-index
                        (schema-print-name schema)
                        (flag-unparse (schema-applicable schema))))
   (applicable-if (fix= 0 (rem schema-index 2))
                  (format *output-stream* "~35T")
                  (format *output-stream* "~%")))
  (applicable-format "~%"))
```

```
;;; *** start update accessibility / create goal-directed actions ***

;;; NOTE: the MIT Press version of Drescher's algorithm doesn't use
;;; any of this code, rather, goal-directed actions are created for
;;; each unique result which is included in a schema

;;; this code updates accessibility for items and conjunctions and
;;; makes goal-directed-actions for those which are highly accessible
;;; must be called *after* schemas have been marked applicable

;;; the following two functions accessible-key and accessible-test are
;;; used via assoc by the main update-accessibility routine -- they
;;; should only be used by that routine (they are fairly specifically
;;; designed for that routine only)

;;; accessible-test assumes that result-schema has a non-nil result
;;; (obviously not interesting), and that context-schema has a non-nil
;;; context (as the first pass in the main update-accessibility insures
;;; that these are all dealt with)

;;; when calling via assoc, result-schema is bound to the result of
;;; evaluating the key for the given car of the assoc pair
;;; (in the case of the vpdate-accessibility routine, the car is a
;;; schema-index, and the accessible-key function gives us the actual
;;; schema)
;;; context-schema is bound to the argument that assoc is looking for
;;; (in the update-accessibility routine, it is the schema for which we
;;; are attempting to find a result which chains to its context)
;;; returns t when result of result-schema implies context of
;;;   context-schema
;;; update accessibility definitions continued
```

accessible-key

```
(defun accessible-key (y)
  (get-schema y))
```

accessible-test

```
(defun accessible-test (context-schema result-schema)
  ;; result must be non-nil to satisfy anything
  (let ((result-data (schema-data result-schema))
        (context-data (schema-data context-schema)))
    (if (schema-data-result-conj-p result-data)
        ;; if result conjunction, check to see if context
        ;; is conjunction too
        (if (schema-data-context-conj-p context-data)
            ;; if so, entry in the result conjunction inclusion
            ;; array set correctly implies context satisfied
            (flag-truep
             (flag-array-get-flag
              (conj-inclusion-array
               (get-conj (schema-result-item result-schema)))
              (schema-context-item context-schema)))
          ;; if context not conjunction, check array directly
          (state-array-included-p
           (conj-item-array
            (get-conj (schema-result-item result-schema)))
           (schema-context-array context-schema)
           25))
      ;; otherwise, context must be single, and stuff match up
      ;; (return the value t or nil as appropriate)
      (and (schema-data-context-single-p context-data)
           (state-eq (if (schema-data-result-negated-p result-data)
                            (make-state-off)
                          (make-state-on))
                        (state-array-get-state
                         (schema-context-array context-schema)
                         (schema-result-item result-schema)))))))
```

truefloat>

```
;; to speed up computation in assoc
(defun truefloat> (x y)
  (> (the short-float x)
     (the short-float y)))
```

accessible-item-pos

```
(defvar *accessible-item-pos*
  (make-flag-array 15))
```

accessible-item-neg

```
(defvar *accessible-item-neg*
  (make-flag-array 15))
```

accessible-conj-pos

```
(defvar *accessible-conj-pos*
  (make-flag-array 10))
```

```
;;; update accessibility definitions continued

;;; description of update-accessibility algorithm:
;;; when a schema is reached through any path, mark it
;;; for the first round, form a list of all applicable schemas with
;;; medium reliability and non-nil results
;;; for every round, keep a list of schemas reached on the previous
;;; round, with corresponding reliability, sorted from highest to
;;; lowest
;;; to form the next round of schemas, go through all schemas,
;;; using find to see if any members of the last round list
;;; have a result which chains to the context of the current schema
;;; if a schema is found which can be chained to,
;;; if marked as visited previously, and reliability higher, then
;;; update old value and put on next round list -- otherwise ignore
;;; if not visited previously and above threshold, put on next round
;;; list
;;; in this fashion, every reachable schema through a path of
;;; accessible schemas can be found and marked
;;; the result item or conjunction for each marked schema is then
;;; marked -- and then each item and conjunction has its accessibility
;;; updated

;;; note: current behavior of program has conjunctive result ->
;;; conjunctive context -- but if a conjunction *includes* another one,
;;; that conjunction isn't marked explicitly accessible (unless it is
;;; accessible through some other path)

;;; accessibility in the negative direction for conjunctions is
;;; definitely not important -- a negated conjunction is a disjunction,
;;; and a disjunctive goal makes no sense within the context of this
;;; mechanism

;;; to avoid unnecessary proliferation of fairly useless goal-directed
;;; actions, this routine was modified to as to not have negated items
;;; as goals either -- this departure from the original paper can be
;;; undone by carefully removing commented out code below and in the
;;; definition of the item datatype
;;; major function: update accessibility

(defun update-accessibility ()
  (flag-array-clear *accessible-item-pos* 15)
; (flag-array-clear *accessible-item-neg* 15)
  (flag-array-clear *accessible-conj-pos* 10)
  (let ((visited nil)
        (old-visited nil))
    (dotimes
     (x *schema-number*)
     (let* ((schema (get-schema x))
            (data (schema-data schema)))
       (if (and (schema-data-applicable-p data)
                (flag-falsep (schema-data-result-empty data))
                (weighted-rate-medium-p (schema-reliability schema)))
           (progn (setf (schema-marked schema) (make-flag-true))
                  (setq visited (acons x (schema-reliability schema) visited)))
         (setf (schema-marked schema) (make-flag-false)))))
    (setq visited (sort visited  #'truefloat> :key #'cdr))
    (setq old-visited (copy-alist visited))
    (accessibility-format "initial visited ~A~%" visited)
    (dotimes
     (y 4)
     (let ((new-visited nil))
       (accessibility-format "pass ~D~%" y)
       (accessibility-format "old-visited ~A~%" old-visited)
       (dotimes
        (x *schema-number*)
        (let* ((schema (get-schema x))
               (data (schema-data schema))
               (reliability (schema-reliability schema)))
          (declare (short-float reliability))
          ;; if empty result, not interesting, and must also have high
          ;; enough reliability (otherwise anything found returned
          ;; would be too low to have new-reliability above the threshold)
          (if (and (flag-falsep (schema-data-result-empty data))
                   (weighted-rate-medium-p reliability))
              ;; the obscure assoc command sends the result of
              ;; applying accessible-key to each element of
              ;; old-visited, along with schema, to accessible-test --
              ;; accessible test is expected to return t
              ;; if one is found where the schema in
              ;; question has a result which includes the context of
              ;; schema
              (let* ((found (assoc schema old-visited
                                   :key #'accessible-key
                                   :test #'accessible-test)))
;;; update accessibility function continued

                (if found
                    (let ((new-reliability
                           (weighted-rate* reliability (cdr found))))
                      (declare (short-float new-reliability))
                      (accessibility-format
                       "schema ~D reliability ~6,4F prior-reliability ~6,4F ~
                        new-reliability ~6,4F~%"
                       x reliability (cdr found) new-reliability)
                      ;; if already marked
                      (if (schema-marked-p schema)
                          ;; check to see if this is a more reliable path
                          (if (float> new-reliability (cdr (assoc x visited)))
                              ;; if so, replace the old value with the new
                              ;; one, and add it to the new-visited array
                              ;; (as the new value may allow certain
                              ;; children which didn't succeed before to
                              ;; succeed this time)
                              (progn
                                (rplacd (assoc x visited) new-reliability)
                                (setq new-visited
                                      (acons x new-reliability new-visited))))
                        ;; otherwise not marked, so mark and add to
                        ;; new-visited array if greater than threshold
                        (if (weighted-rate-medium-p new-reliability)
                            (progn
                              (setf (schema-marked schema) (make-flag-true))
                              (setq new-visited
                                    (acons x new-reliability new-visited)
                                    visited
                                    (acons x new-reliability visited))))))))))
       (if (not new-visited)
           (return))
       ;; at this point, new-visited has what should be used for
       ;; old-visited on the next iteration
       (setq new-visited (sort new-visited #'truefloat> :key #'cdr))
       (setq old-visited new-visited)
       (accessibility-format "visited ~A~%" visited))))
```

```
;;; update accessibility function continued

  ;; at this point, all "accessible" schemas should be marked, so go
  ;; through them, if result conjunctive, mark the included
  ;; conjunctions (which includes the result conjunction)
  ;; if not, just mark the item(s)
  (dotimes
   (x *schema-number*)
   (let* ((schema (get-schema x))
          (data (schema-data schema)))
     (if (and (schema-data-marked-p data)
              (flag-falsep (schema-data-result-negated data)))
         (let ((result-item (schema-result-item schema)))
           (if (schema-data-result-conj-p data)
               (flag-array-ior (conj-inclusion-array
                                (get-conj result-item))
                               *accessible-conj-pos*
                               10)
             (setf (flag-array-get-flag
                    *accessible--item-pos*
                    result-item)
                   (make-flag-true)))))))
; old version for negated results (note: doesn't do inclusive
; conjunctions right)
; (dotimes
;  (x *schema-number*)
;  (let* ((schema (get-schema x))
;         (data (schema-data schema)))
;    (if (schema-data-marked-p data)
;      (let ((result-item (schema-result-item schema)))
;        (if (schema-data-result-conj-p data)
;            (if (flag-falsep (schema-data-result-negated data))
;                (setf (flag-array-get-flag
;                        *accessible-conj-pos*
;                        result-item)
;                       (make-flag-truc)))
;          (setf (flag-array-get-flag
;                 (if (flag-truep (schema-result-negated schema))
;                     *accessible-item-neg*
;                   *accessible-item-pos*)
;                 result-item)
;                (make-flag-true)))))))
```

```
;;; update accessibility function continued

  ;; actually modify the accessibility numbers for the items and conjunctions
  ;; for each marked conjunction, increase accessibility and mark the
  ;; included (positive) items
  (accessibility-format "updating accessibility~%")
  (dotimes
   (x *conj-number*)
   (let ((conj (get-conj x)))
     (if (flag-truep (flag-array-get-flag *accessible-conj-pos* x))
         (progn
           (conj-acc-pos-update conj t)
           (flag-array-ior (conj-pos-flag-array conj)
                           *accessible-item-pos* 10)
           (accessibility-format "~A~%" conj)
           (if (and (flag-falsep (conj-gd-pos-created conj))
                    (conj-acc-pos-high-p conj))
               (progn
                 (setf (conj-gd-pos-created conj) (make-flag-true))
                 (make-gd-action-conj-pos x))))
       (progn
         (conj-acc-pos-update conj nil)
         (accessibility-format "~A~%" conj)))))
  (dotimes
   (x *item-number*)
   (let ((item (get-item x)))
     (item-acc-pos-update
      item
      (flag-truep (flag-array-get-flag *accessible-item-pos* x)))
;    (item-acc-neg--update
;     item
;     (flag-truep (flag-array-get-flag *accessible-item-neg* x)))
     (accessibility-format "~A~%" item)
     (if (and (flag-falsep (item-gd-pos-created item))
              (item-acc-pos-high-p item))
         (progn
           (setf (item-gd-pos-created item) (make-flag-true))
           (make-gd-action-item-pos x)))
;    (if (and (flag-falsep (item-gd-neg-created item))
;             (item-acc-neg-high-p item))
;      (progn
;       (setf (item-gd-neg-created item) (make-flag-true))
;       (make-gd-action-item-neg x)))
     )))
```

```
;;; create goal-directed action functions

(defun make-gd-action-item-pos (item-index)        make-gd-action-item-pos
  (main-format "~5D goal-directed-action-item-pos-created ~D ~A~%"
               *clock-tick*
               item-index
               (item-print-name (get-item item-index))))

;(defun make-gd-action-item-neg (item-index)       make-gd-action-item-neg
; (main-format "~5D goal-directed-action-item-neg-created ~D ~A~%"
;              *clock-tick*
;              item-index
;              (item-print-name (get-item item-index))))

(defun make-gd-action-conj-pos (conj-index)        make-gd-action-conj-pos
  (main-format "~5D goal-directed-action-conj-pos-created ~D ~A~%"
               *clock-tick*
               conj-index
               (conj-print-name (get-conj conj-index))))

;;; *** end update accessibility / create goal-directed actions code ***
;;; major functions: item-update-state and conj-update-state
```

item-update-state

```
(defun item-update-state ()
  (dotimes
   (x *item-number*)
   (let ((current-item (get-item x)))
     (if (item-syn-item-p current-item)
         (setf
          ;; last state (post) = current state (pre)
          (item-last-state current-item)
          (item-current-state current-item)
          ;; put "both" into current-state and microworld-state
          ;; "both" is 11 and inclusive ORs with anything
          ;; written into each synthetic item state while determining
          ;; the state of the primitive items
          (item-current-state current-item)
          (make-state-both)
          (get-microworld-state x)
          (make-state-both))
       (let ((new-state (state-parse (funcall (item-code current-item)))))
         (setf
          ;; last state (post) = current state (pre)
          (item-last-state current-item)
          (item-current-state current-item)
          ;; current state (post) = result of calling code
          (item-current-state current-item)
          new-state
          (get-microworld-state x)
          new-state)
         ;; update generality -- increment rate if state is On
         (item-generality-update current-item (state-on-p new-state)))))))
```

conj-update-state

```
(defun conj-update-state ()
  (dotimes
   (x *conj-number*)
   (let* ((current-conj (get-conj x))
          (new-state (state-parse
                      (state-array-included-p
                       *microworld-state*
                       (conj-item-array current-conj)
                       25))))
     (setf
      ;; last state (post) = current state (pre)
      (conj-last-state current-conj)
      (conj-current-state current-conj)
      ;; current state (post) = result of calling code
      (conj-current-state current-conj)
      new-state))))
```

;;; major function: schema–update–activated

;;; if schema has same action, is not a goal–directed–action schema,
;;; and is applicable (i.e. context–satisfied) then is activated
;;; otherwise, if goal–directed–action and result satisfied, mark activated

schema-update-activated

```
(defun schema-update-activated (action-index)
  (activated-format "updating activated~%")
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (setf (schema-activated schema)
           (flag-parse
            (and (fix= (schema-action-item schema) action-index)
                 (if (schema-action-gd-p schema)
                     (schema-result-satisfied-p schema)
                   (schema-applicable-p schema)))))
     (activated-format "~4D ~25A~A"
                       schema-index
                       (schema-print-name schema)
                       (flag-unparse (schema-activated schema))))
   (activated-if (fix= 0 (rem schema-index 2))
                 (format *output-stream* "~35T")
                 (format *output-stream* "~%")))
  (activated-format "~%"))
```

```
(defvar *reliable-item-pos* (make-flag-array 15))          *reliable-item-pos*
(defvar *reliable-item-neg* (make-flag-array 15))          *reliable-item-neg*
(defvar *reliable-conj* (make-flag-array 10))                  *reliable-conj*

(defun schema-not-context-overridden-p schema-not-context-overridden-p
  (schema)

  (let ((record-offset 0)
        (array-index 0)
        (ext-context (schema-extended-context schema)))
    ;; iterate through all items
    (dotimes
     (x *item-number* t)
     ;; for each, check to see if the counter value is 13 or higher
     (if (fix< 12
               (counter-array-value
                ext-context array-index record-offset))
         ;; if so, check to see if counter positive and item Off
         ;; or counter negative and item On
         ;; if either is true, overridden
         (if (flag-truep
              (counter-array-pos
               ext-context array-index record-offset))
             (if (state-off-p (get-microworld-state x))
                 (return nil))
           (if (state-on-p (get-microworld-state x))
               (return nil))))
     (if (fix= *counter-record-max-offset* record-offset)
         (setq record-offset 0
               array-index (fix1+ array-index))
       (setq record-offset (fix+ *counter-bits* record-offset))))))

;;; when the current state is KNOWN, put in current-state
;;; when it is just guessed, put in maybe-state
;;; set-time is updated when current-state is changed, this can easily
;;; be used to check and make sure an earlier (higher precedent) value
;;; isn't being clobbered

(defun syn-item-update-state ()                         syn-item-update-state
  (dotimes
   (x *syn-item-number*)
   (setf (syn-item-maybe-state (get-syn-item x))
         (make-state-unknown)))
  (schema-update-host-results)
  (syn-item-update-phase-one)
  (syn-item-update-phase-two-a)
  (syn-item-update-phase-three)
  (syn-item-update-phase-two-b)
  (syn-item-update-phase-four)
  (syn-item-update-phase-five))
```

schema-update-host-results

```
(defun schema-update-host-results ()
  (result-format "updating host schema results~%")
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (if (schema-syn-item-p schema)
         (setf (schema-result-satisfied schema)
               (flag-parse
                (or (schema-result-empty-p schema)
                    (if (schema-result-conj-p schema)
                        (conj-satisfied-p
                         (schema-result-item schema))
                      (or (state-eq
                           (if (schema-result-negated-p schema)
                               (make-state-off)
                             (make-state-on))
                           (get-microworld-state
                            (schema-result-item schema)))
                          (state-eq
                           (make-state-both)
                           (get-microworld-state
                            (schema-result-item schema)))))))))
    (result-format "~4D ~25A~A"
                   schema-index
                   (schema-print-name schema)
                   (flag-unparse (schema-result-satisfied schema))))
  (result-if (fix= 0 (rem schema-index 2))
             (format *output-stream* "~35T")
             (format *output-stream* "~%")))
 (result-format "~%"))
```

```
;;; syn-item-update-state functions continued
```

syn-item-update-phase-one

```
(defun syn-item-update-phase-one ()

  ;; phase one
  ;; if host schema activated and not overridden
  ;; result obtains/does not obtain -> On/Off
  ;; these go into current-state as they are not to be overridden
  ;; (for efficiency, part of phase two is also done: if host is
  ;; overridden, put Off as maybe-state)

  (dotimes
   (x *syn-item-number*)
   (let* ((syn (get-syn-item x))
          (schema (get-schema (syn-item-host-schema syn))))
     (if (schema-activated-p schema)
         (if (schema-not-context-overridden-p schema)
             (if (schema-result-satisfied-p schema)
                 (setf (syn-item-current-state syn)
                       (make-state-on)
                       (syn-item-unknown-time syn)
                       (fix+ *clock-tick*
                             (average-value
                              (syn-item-on-duration syn)))
                       (syn-item-set-time syn)
                       *clock-tick*)
               (setf (syn-item-current-state syn)
                     (make-state-off)
                     (syn-item-unknown-time syn)
                     (fix+ *clock-tick*
                           (average-value
                            (syn-item-off-duration syn)))
                     (syn-item-set-time syn) *clock-tick*))
           (setf (syn-item-maybe-state syn) (make-state-off)))))))
```

syn-item-update-phase-two-a

```
(defun syn-item-update-phase-two-a ()

  ;; NOTE: the accidental clobbering of phase one values can be
  ;; prevented now by checking set-tick ---> if equal to *clock-tick*,
  ;; it has already been set and should not be further modified

  ;; phase two
  ;; overriden host schema -> Off in maybe-state (done in phase one)
  ;; not overridden host schema with reliable applicable child
  ;;    gives On in maybe-state (unless maybe-state set in phase one)

  (dotimes
   (x *schema-number*)
   (let* ((schema (get-schema x))
          (data (schema-data schema)))
     (if (flag-falsep (schema-data-result-empty data))
         (let ((parent (get-schema (schema-parent schema))))
           (if (and
                 (schema-syn-item-p parent)
                 (schema-data-applicable-p data)
                 (weighted-rate-high-p (schema-reliability schema))
                 (state-unknown-p
                  (syn-item-maybe-state (schema-reifier parent)))
                 (schema-not-context-overridden-p schema))
               (setf (syn-item-maybe-state (schema-reifier parent))
                     (make-state-on))))))))
```

syn-item-update-phase-three

```
(defun syn-item-update-phase-three ()
  ;; phase three
  ;; reliable activated schemas with synthetic items in the result
  ;; indicate that the item should be turned On/Off
  ;; move phase two and phase three results up into current-state
  ;; if not blocked by already being set this time around

  (dotimes
   (x *schema-number*)
   (let* ((schema (get-schema x))
          (data (schema-data schema)))
     (if (and
          (schema-data-activated-p data)
          (weighted-rate-high-p (schema-reliability schema))
          (flag-falsep (schema-data-result-empty data)))
         (let ((result-item (schema-result-item schema)))
           (if (schema-data-result-conj-p data)
               (flag-array-ior (conj-inclusion-array
                                (get-conj result-item))
                               *reliable-conj*
                               10)
             (setf (flag-array-get-flag
                    (if (schema-data-result-negated-p data)
                        *reliable-item-neg*
                      *reliable-item-pos*)
                    result-item)
                   (make-flag-true)))))))
  (dotimes
   (x *conj-number*)
   (if (flag-truep (flag-array-get-flag *reliable-conj* x))
       (let ((conj (get-conj x)))
         (flag-array-ior (conj-pos-flag-array conj)
                         *reliable-item-pos* 10)
         (flag-array-ior (conj-neg-flag-array conj)
                         *reliable-item-neg* 10))))
```

```
;;; syn-item-update-phase-three continued

(dotimes
 (x *item-number*)
 (let* ((item (get-item x))
       (data (item-data item))}
   (if (item-data-syn-item-p data)
       (let ((neg (flag-array-get-flag *reliable-item-neg* x))
             (pos (flag-array-get-flag *reliable-item-pos* x))
             (syn (get-syn-item (item-syn-item-index item))))
         (if (fix/= *clock-tick* (syn-item-set-time syn))
             (if (flag-truep neg)
                 (if (flag-truep pos)
                     ;; if both are true, set unknown
                     (setf (syn-item-current-state syn)
                          (make-state-unknown)
                          (syn-item-set-time syn)
                          *clock-tick*)
                   ;; neg true, pos false, if not marked in phase
                   ;; two, set off
                   (if (state-unknown-p (syn-item-maybe-state syn))
                       (setf (syn-item-current-state syn)
                            (make-state-off)
                            (syn-item-unknown-time syn)
                            (fix+ *clock-tick*
                                 (average-value
                                  (syn-item-off-duration syn)))
                            (syn-item-set-time syn)
                            *clock-tick*)
                     ;; marked in phase two, set unknown
                     (setf (syn-item-current-state syn)
                          (make-state-unknown)
                          (syn-item-set-time syn)
                          *clock-tick*)))
               ;; neg false
               (if (flag-truep pos)
                   ;; neg false, pos true, if not marked in phase
                   ;; two, set on
                   (if (state-unknown-p (syn-item-maybe-state syn))
                       (setf (syn-item-current-state syn)
                            (make-state-on)
                            (syn-item-unknown-time syn)
                            (fix+ *clock-tick*
                                 (average-value
                                  (syn-item-on-duration syn)))
                            (syn-item-set-time syn)
                            *clock-tick*)
                     ;; marked in phase two, set unknown
                     (setf (syn-item-current-state syn)
                          (make-state-unknown)
                          (syn-item-set-time syn)
                          *clock-tick*)))))))))
```

```
(defun syn-item-update-phase-two-b ()
```

syn-item-update-phase-two-b

```
  ;; phase two revisited
  ;; if phase two and three conflicted, the current-state was set to
  ;; unknown —— any synthetic item which hasn't been set this cycle
  ;; and has a non-unknown maybe-state should use the maybe--state to
  ;; update current-state

  (dotimes
   (x *syn-item-number*)
   (let ((syn (get-syn-item x)))
     (if (and (fix/= *clock-tick* (syn-item-set-time syn))
              (state-noteq (make-state-unknown)
                           (syn-item-maybe-state syn)))
         (setf (syn-item-current-state syn)
               (syn-item-maybe-state syn)
               (syn-item-unknown-time syn)
               (fix+ *clock-tick*
                     (average-value
                      (if (state-on-p
                           (syn-item-maybe-state syn))
                          (syn--item-on-duration syn)
                        (syn-item-off-duration syn))))
               (syn-item-set-time syn)
               *clock-tick*)))))
```

```
(defun syn-item-update-phase-four ()
```

syn-item-update-phase-four

```
  ;; phase four
  ;; timeouts

  (dotimes
   (x *syn-item-number*)
   (let ((syn (get-syn-item x)))
     (if (and (fix/= *clock-tick* (syn-item-set-time syn))
              (fix= *clock-tick* (syn-item-unknown-time syn)))
         (setf (syn-item-current-state syn)
               (make-state-unknown)
               (syn-item-set-time syn)
               *clock-tick*)))))
```

```
;;; syn-item-update-state functions continued

(defun syn-item-update-phase-five ()
```

syn-item-update-phase-five

```
  ;; phase five
  ;; update the items and microworld-state array with the newly
  ;; calculated information -- also update generality for the
  ;; synthetic items

  (dotimes
   (x *item-number*)
   (let ((current-item (get-item x)))
     (if (item-syn-item-p current-item)
         (let ((new-state
                (syn-item-current-state
                 (get-syn-item (item-syn-item-index current-item)))))
           (setf (item-current-state current-item) new-state
                 (get-microworld-state x) new-state)
           (item-generality-update current-item (state-on-p new-state)))))))
```

```
;;; major functions:
;;; schema-update-all-results and schema-update-reliability
```

schema-update-all-results

```
(defun schema-update-all-results ()
  (result-format "updating all results~%")
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (setf (schema-result-satisfied schema)
           (flag-parse
            (or (schema-result-empty-p schema)
                (if (schema-result-conj-p schema)
                    (conj-satisfied-p
                     (schema-result-item schema))
                  (state-eq
                   (if (schema-result-negated-p schema)
                       (make-state-off)
                     (make-state-on))
                   (get-microworld-state (schema-result-item schema)))))))
     (result-format "~4D ~25A~A"
                    schema-index
                    (schema-print-name schema)
                    (flag-unparse (schema-result-satisfied schema))))
   (result-if (fix= 0 (rem schema-index 2))
              (format *output-stream* "~35T")
              (format *output-stream* "~%")))
  (result-format "~%"))
```

schema-update-reliability

```
(defun schema-update-reliability ()
  (reliability-format "updating reliability~%")
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (if (schema-activated-p schema)
         (weighted-rate-update (schema-reliability schema)
                               (schema-result-satisfied-p schema)))
     (reliability-format "~4D ~6,4F"
                         schema-index
                         (schema-reliability schema)))
   (reliability-if (fix= 0 (rem schema-index 2))
                   (format *output-stream* "~35T")
                   (format *output-stream* "~%")))
  (reliability-format "~%"))
```

```
;;; predicted-results definitions
```

predicted-results

```
(defvar *predicted-results*
  (make-state-array 25))
```

predicted-result-conjs

```
(defvar *predicted-result-conjs*
  (make-state-array 20))
```

predicted-result

```
(defmacro predicted-result (item)
  `(state-noteq (make-state-unknown)
                (state-array-get-state *predicted-results* ,item)))
```

predicted-result-conj

```
(defmacro predicted-result-conj (conj)
  `(state-noteq (make-state-unknown)
                (state-array-get-state *predicted-result-conjs*
                                       ,conj)))
```

```
;;; major functions:
;;; update-predicted-results and print-predicted-results
```

update-predicted-results

```
(defun update-predicted-results ()
  (state-array-clear *predicted-results* 25)
  (state-array-clear *predicted-result-conjs* 20)
  (dotimes
   (schema-index *schema-number*)
   (let ((schema (get-schema schema-index)))
     (if (and (flag-falsep (schema-result-empty schema))
              (weighted-rate-high-p (schema-reliability schema))
              (schema-activated-p schema)
              (schema-result-satisfied-p schema))
         (if (schema-result-conj-p schema)
             (progn
               (setf (state-array-get-state *predicted-result-conjs*
                                            (schema-result-item schema))
                     (if (schema-result-negated-p schema)
                         (make-state-off)
                       (make-state-on)))
               (state-array-ior
                (conj-item-array
                 (get-conj (schema-result-item schema)))
                *predicted-results*
                25))
           (setf (state-array-get-state *predicted-results*
                                        (schema-result-item schema))
                 (if (schema-result-negated-p schema)
                     (make-state-off)
                   (make-state-on))))))))
```

print-predicted-results

```
(defmacro print-predicted-results (&rest args)
  (if *predicted-results-enabled*
      '(print-predicted-results-enabled ,@args)))
```

print-predicted-results-enabled

```
(defun print-predicted-results-enabled ()
  (predicted-results-format "predicted-results~%")
  (dotimes
   (x *item-number*)
   (let ((foo (state-array-get-state *predicted-results* x)))
     (cond ((state-unknown-p foo) nil)
           (t (predicted-results-format
               "~A~%"
               (item-print-name (get-item x)))))))
    (predicted-results-format "predicted-result-conjunctions~%")
  (dotimes
   (x *conj-number*)
   (let ((foo (state-array-get-state *predicted-result-conjs* x)))
     (cond ((state-unknown-p foo) nil)
           (t (predicted-results-format
               "~A~%"
               (conj-print-name (get-conj x))))))))
```

;;; major function: schema-update-ext-item-stats

schema-update-ext-item-stats

```
(defun schema-update-ext-item-stats ()
  (dotimes
   (item-index *item-number*)
     (let ((current-item (get-item item-index))
           (record-offset 0))
       (multiple-value-bind
        (array-index record-position)
        (get-counter-array-index item-index)
        (setq record-offset (counter-record-offset record-position))
        (item-format "~A~%" current-item)
     (dotimes
      (schema-index *schema-number*)
      (let ((schema (get-schema schema-index)))
        (if (schema-result-empty-p schema)
            (let ((activated (schema-activated schema))
                  (positive (schema-extended-result-pos schema))
                  (negative (schema-extended-result-neg schema))
                  (current-state (item-current-state current-item))
                  (last-state (item-last-state current-item)))
             (ext-stats-format "~4D res " schema-index)
             ;;; do positive transition -- activation matches and old is off
             (cond ((and (state-off-p last-state)
                         (not (and
                               (flag-falsep activated)
                               (predicted-result item-index)))
                         (flag-eq
                          (counter-array-toggle
                           positive array-index record-offset)
                          activated))
                    (ext-stats-format "+t ")
                    ;;; check to see if new is on
                    (if (state-on-p current-state)
                        (progn
                          (ext-stats-format "~A->"
                                            (counter-unparse-from-array
                                             positive array-index record-offset))
                          (counter-array-toggle-toggle
                           positive array-index record-offset)
                          (counter-array-modify-value
                           positive array-index record-offset activated)
                          (ext-stats-format "~A"
                                            (counter-unparse-from-array
                                             positive array-index record-offset)))
                      (progn
                        (ext-stats-format "~A->"
                                          (counter-unparse-from-array
                                           positive array-index record-offset))
                        (counter-array-toggle-toggle
                         positive array-index record-offset)
                        (ext-stats-format "~A"
                                          (counter-unparse-from-array
                                           positive array-index record-offset))))
;;; schema-update-ext-item-stats continued

;;; do negative transition -- old is on
((and (state-on-p last-state)
      (not (and (flag-falsep activated)
                (predicted-result item-index)))
      (flag-eq
       (counter-array-toggle
        negative array-index record-offset)
       activated))
 (ext-stats-format "-t ")
 ;;; check to see if new is off
 (if (state-off-p current-state)
     (progn
       (ext-stats-format "~A->"
                   (counter-unparse-from-array
                    negative array-index record-offset))
       (counter-array-toggle-toggle
        negative array-index record-offset)
       (counter-array-modify-value
        negative array-index record-offset activated)
       (ext-stats-format "~A"
                   (counter-unparse-from-array
                    negative array-index record-offset)))
   (progn
     (ext-stats-format "~A->"
                 (counter-unparse-from-array
                  negative array-index record-offset))
     (counter-array-toggle-toggle
      negative array-index record-offset)
     (ext-stats-format "~A"
                 (counter-unparse-from-array
                  negative array-index record-offset)))))
(t (ext-stats-format "no "))))
```

```
;;; schema-update-ext-item-stats continued

          ;; non-empty result so update extended-context
          (progn
            (ext-stats-format "~4D con " schema-index)
            (if (and (schema-activated-p schema)
                    (or
                     (state-unknown-p
                      (state-array-get-state
                       (schema-context-children schema) item-index))
                     (state-noteq
                      (state-array-get-state
                       (schema-context-children schema) item-index)
                      (item-last-state current-item))))
               (let* ((ext-context (schema-extended-context schema))
                      (last-state (item-last-state current-item))
                      (toggle (counter-array-toggle
                               ext-context array-index record-offset)))
                 (if (or (and (flag-truep toggle)
                              (state-on-p last-state))
                         (and (flag-falsep toggle)
                              (state-off-p last-state)))
                     (progn
                       (ext-stats-if
                        (state-on-p last-state)
                        (ext-stats-format "+t ")
                        (ext-stats-format "-t "))
                       (ext-stats-format
                        "~A->"
                        (counter-unparse-from-array
                         ext-context array-index record-offset))
                       (counter-array-toggle-toggle
                        ext-context array-index record-offset)
                       (if (schema-result-satisfied-p schema)
                           (counter-array-modify-value
                            ext-context array-index record-offset
                            (flag-parse (state-on-p last-state))))
                       (ext-stats-format
                        "~A "
                        (counter-unparse-from-array
                         ext-context array-index record-offset)))
                   (ext-stats-format "no match state/toggle")))
             (ext-stats-format "not activated or deferred"))))
       (ext-stats-if (fix= 0 (rem schema-index 2))
                           (format *output-stream* "~35T")
                           (format *output-stream* "~%"))))
    (ext-stats-format "~%")))))
```

;;; major function: schema–update–ext–conj–stats

schema-update-ext-conj-stats

```
(defun schema-update-ext-conj-stats ()
  (dotimes
   (conj-index *conj-number*)
   (let* ((current-conj (get-conj conj-index))
          (record-offset 0)
          (current-state (conj-current-state current-conj))
          (last-state (conj-last-state current-conj)))
     (multiple-value-bind
      (array-index record-position)
      (get-counter-array-index conj-index)
      (setq record-offset (counter-record-offset record-position))
      (conj-format "~A~%" current-conj)
      (dotimes
       (schema-index *schema-number*)
       (let ((schema (get-schema schema-index)))
         (ext-conj-stats-format "~4D res conj " schema-index)
         (if (schema-result-empty-p schema)
             (let ((activated (schema-activated schema))
                   (positive (schema-extended-result-conj-pos schema)))
               ;;; do positive transition -- activation matches and old is off
               (cond ((and (state-off-p last-state)
                           (not (and
                                 (flag-falsep activated)
                                 (predicted-result-conj conj-index)))
                           (flag-eq
                            (counter-array-toggle
                             positive array-index record-offset)
                            activated))
                      (ext-conj-stats-format "+t ")
```

```
                  ;;; check to see if new is on
                  (if (state-on-p current-state)
                      (progn
                        (ext-conj-stats-format
                         "~A->" (counter-unparse-from-array
                                positive array-index record-offset))
                        (counter-array-toggle-toggle
                         positive array-index record-offset)
                        (counter-array-modify-value
                         positive array-index record-offset activated)
                        (ext-conj-stats-format
                         "~A" (counter-unparse-from-array
                              positive array-index record-offset)))
                    (progn
                      (ext-conj-stats-format
                       "~A->" (counter-unparse-from-array
                              positive array-index record-offset))
                      (counter-array-toggle-toggle
                       positive array-index record-offset)
                      (ext-conj-stats-format
                       "~A" (counter-unparse-from-array
                            positive array-index record-offset)))))
                 (t (ext-conj-stats-format "no "))))
        (ext-conj-stats-format "-- res non-empty")))
    (ext-conj-stats-if (fix= 0 (rem schema-index 2))
                           (format *output-stream* "~35T")
                           (format *output-stream* "~%")))))
(ext-conj-stats-format "~%")))
```

```
;;; major function: maybe-spinoff-schema

(defun maybe-spinoff-schema ()
  (dotimes
   (schema-index *schema-number*)
   (when
   (let ((schema (get-schema schema-index))
         (array-index 0)
         (record-offset 0))
     (if (schema-result-empty-p schema)
         (let ((result-pos
                (schema-extended-result-pos schema))
               (result-neg
                (schema--extended-result-neg schema))
               (result-pos-conj
                (schema-extended-result-conj-pos schema))
               (children (schema-result-children schema))
               (conj-children (schema-result-conj-children schema)))
           (dotimes
            (item-index *item-number* nil)
              (if (state-unknown-p
                    (state-array-get-state children item-index))
                  (cond ((and (fix= *counter-maximum*
                                    (counter-array-value
                                     result-pos array-index record-offset))
                              (flag-truep
                               (counter-array-pos
                                result-pos array-index record-offset)))
                         (return
                          (make-spinoff-result schema-index
                                               item-index
                                               (make-state-on))))
                        ((and (fix= *counter-maximum*
                                    (counter-array-value
                                     result-neg array-index record-offset))
                              (flag-truep
                               (counter-array-pos
                                result-neg array-index record-offset)))
                         (return
                          (make-spinoff-result schema-index
                                               item-index
                                               (make-state-off))))
                        (t nil)))
;;; maybe-spinoff-schema continued

              (if (and (fix< item-index *conj-number*)
                      (flag-falsep
                       (flag-array-get-flag conj-children item-index)))
                  (cond ((and
                           (fix= *counter-maximum*
                                 (counter-array-value
                                  result-pos-conj array-index record-offset))
                           (flag-truep
                            (counter-array-pos
                             result-pos-conj array-index record-offset)))
                          (return
                           (make-spinoff-result-conj
                            schema-index
                            item-index)))
                       (t nil)))
             (if (fix= *counter-record-max-offset* record-offset)
                 (setq record-offset 0
                       array-index (fix1+ array-index))
               (setq record-offset (fix+ *counter-bits* record-offset)))))
       ;; non-empty result so check for context spinoffs
       (let ((context (schema-extended-context schema))
             (children (schema-context-children schema))
             (current-item -1)
             (current-state (make-state-unknown)))
         (dotimes
            (item-index *item-number*
                       (if (state-noteq (make-state-unknown) current-state)
                           (make-spinoff-context schema-index
                                                 current-item
                                                 current-state)
                         nil))
           (if (and
                (state-unknown-p
                 (state-array-get-state children item-index))
                (fix= *counter-maximum*
                      (counter-array-value
                       context array-index record-offset))
                (or (state-unknown-p current-state)
                    (item-generality-< (get-item item-index)
                                       (get-item current-item))))
               (setq current-state
                     (if (flag-truep
                          (counter-array-pos
                           context array-index record-offset))
                         (make-state-on)
                       (make-state-off))
                     current-item item-index))
           (if (fix= *counter-record-max-offset* record-offset)
               (setq record-offset 0
                     array-index (fix1+ array-index))
             (setq record-offset (fix+ *counter-bits* record-offset))))))
  (return 'maybe-spinoff-schema-finished))))
```

```
;;; major function: make-spinoff-result

;;; as result spinoffs can only occur from empty-result schemas, and
;;; empty result schemas must necessarily have empty contexts, this
;;; routine does not have to deal with anything but spinning off from
;;; "blank" schemas -- i.e. those which do not have any context or
;;; result -- remember, all flags are false by default
```

make-spinoff-result

```
(defun make-spinoff-result (schema item state)
 (let* ((parent-schema (get-schema schema))
        (spinoff-schema (make-action-schema
                         (schema-action-item parent-schema))))
   (setf (schema-result-item spinoff-schema)
         item
         (schema-context-empty spinoff-schema)
         (make-flag-true)
         (schema-parent spinoff-schema)
         schema)
   (if (state-off-p state)
       (setf (schema-result-negated spinoff-schema)
             (make-flag-true)))
   (schema-update-print-name spinoff-schema)
   (main-format "~5D spinoff-result  ~A~6A ~A -> ~A~%"
                *clock-tick*
                (state-unparse state)
                (item-print-name (get-item item))
                (schema-print-name parent-schema)
                (schema-print-name spinoff-schema))
   (setf
    (state-array-get-state
     (schema-result-children parent-schema) item)
    state)
   spinoff-schema))
```

```
;;; major function: make-spinoff-result-conj

(defun make-spinoff-result-conj (schema conj)
  (let* ((parent-schema (get-schema schema))
         (spinoff-schema (make-action-schema
                         (schema-action-item parent-schema))))
    (setf (schema-result-item spinoff-schema)
          conj
          (schema-result-conj spinoff-schema)
          (make-flag-true)
          (schema-context-empty spinoff-schema)
          (make-flag-true)
          (schema-parent spinoff-schema)
          schema)
    (schema-update-print-name spinoff-schema)
    (main-format "~5D spinoff-result-conjunction  ~A~6A ~A -> ~A~%"
                *clock-tick*
                (state-unparse (make-state-on))
                (conj-print-name (get-conj conj))
                (schema-print-name parent-schema)
                (schema-print-name spinoff-schema))
    (setf
     (flag-array-get-flag
      (schema-result-conj-children parent-schema) conj)
     (make-flag-true))
    spinoff-schema))
```

```
;;; major function: make-spinoff-context

;;; a context spinoff has same context as its parent, but with a new
;;; item added -- note that this invalidates any conjunction
;;; information, so this isn't copied (by default context-conjunction
;;; is false, and context-item is -1)
;;; result and action are direct copies from the parent
;;; the extended-context counters for the parent schema are also reset
;;;   (as required for deferring to a more specific schema)
;;; to inhibit context spinoffs of items which are already in the
;;; context, the context-array is copied to the context-children array
```

make-spinoff-context

```
(defun make-spinoff-context (schema item state)
  (let* ((parent-schema (get-schema schema))
         (spinoff-schema (make-action-schema
                         (schema-action-item parent-schema)))
         (parent-schema-data (schema-data parent-schema)))
    (if (schema-data-context-empty-p parent-schema-data)
        (setf (schema-context-single spinoff-schema) (make-flag-true))
      (state-array-copy (schema-context-array parent-schema)
                        (schema-context-array spinoff-schema)
                        25))
    (setf (state-array-get-state
           (schema-context-array spinoff-schema) item)
          state
          (schema-parent spinoff-schema)
          schema)
    (state-array-copy (schema-context-array spinoff-schema)
                      (schema-context-children spinoff-schema)
                      25)
    (if (schema-data-result-empty-p parent-schema-data)
        (setf (schema-result-empty spinoff-schema) (make-flag-true))
      (setf (schema-result-item spinoff-schema)
            (schema-result-item parent-schema)
            (schema-result-conj spinoff-schema)
            (schema-data-result-conj parent-schema-data)
            (schema-result-negated spinoff-schema)
            (schema-data-result-negated parent-schema-data)))
    (setf (state-array-get-state
           (schema-context-children parent-schema) item)
          state
          (schema-extended-context parent-schema)
          (make-counter-array 75))
    (if (schema-data-action-gd-p parent-schema-data)
        (setf (schema-extended-context-post parent-schema)
              (make-counter-array 75)))
    (schema-update-print-name spinoff-schema)
    (main-format "~5D spinoff-context  ~A~6A ~A -> ~A~%"
                 *clock-tick*
                 (state-unparse state)
                 (item-print-name (get-item item))
                 (schema-print-name parent-schema)
                 (schema-print-name spinoff-schema))
   spinoff-schema))
```

maybe-make-conjs

```
;;; major function: maybe-make-conjs

(defun maybe-make-conjs ()
  (conj-format "making necessary conjunctions...~%")
  (dotimes
   (x *schema-number*)
   (let* ((schema (get-schema x))
          (data (schema-data schema)))
     (if (and (flag-falsep (schema-data-context-single data))
              (flag-falsep (schema-data-context-empty data))
              (weighted-rate-high-p (schema-reliability schema))
              (flag-falsep (schema-data-context-conj data)))
         (progn
           (setf (schema-context-item schema)
                 (make-conj (schema-context-array schema))
                 (schema-context-conj schema)
                 (make-flag-true))
           (schema-update-print-name schema)
           (conj-format "modifying ~4D ~A conj ~D~%"
                        x
                        (schema-print-name schema)
                        (schema-context-item schema))))))
  (dotimes
   (x *conj-number*)
   (conj-format "~A~%" (get-conj x))))
```

```
;;; major function: maybe-make-syn-items

;;; a schema succeeds if, when activated, its result obtains
;;; initially, a schema updates the lc-consy rate, if it
;;; succeeded this time -- then check to see if succeeded last time
;;; and update rate accordingly
;;; when highly lcly-cons and not reliable, stop updating
;;; lc-cons and start updating durations
;;; on-duration : the duration from the first successful execution to
;;; the first unsuccessful one
;;; off-duration : the duration from the first unsuccessful execution to
;;; the first successful one
```

maybe-make-syn-items

```
(defun maybe-make-syn-items ()
  (syn-item-format
   "making synthetic items and updating duration/consistency...~%")
  (dotimes
   (schema-index *schema-number*)
   (syn-item-format "~4D " schema-index)
   (let ((schema (get-schema schema-index)))
     (if (schema-lcly-cons-p schema)
         (if (schema-activated-p schema)
             (if (and (schema-result-satisfied-p schema)
                      (not (schema-succeeded-last-p schema)))
                 (progn
                  (syn-item-format "syn update off-duration before ~3D "
                                   (average-value
                                    (syn-item-off-duration
                                     (get-syn-item
                                      (schema-reifier schema)))))
                  (average-update
                   (syn-item-off-duration
                    (get-syn-item (schema-reifier schema)))
                   (fix- *clock-tick* (schema-first-tick schema)))
                  (syn-item-format "after ~3D "
                                   (average-value
                                    (syn-item-off-duration
                                     (get-syn-item
                                      (schema-reifier schema)))))
                  (syn-item-format "first-tick set ~5D" *clock-tick*)
                  (setf (schema-first-tick schema) *clock-tick*))
;;; maybe-make-syn-items continued
               (if (and (not (schema-result-satisfied-p schema))
                        (schema-succeeded-last-p schema))
                   (progn
                     (syn-item-format "syn update on-duration before ~3D "
                                             (average-value
                                              (syn-item-on-duration
                                               (get-syn-item
                                                (schema-reifier schema)))))
                     (average-update
                      (syn-item-on-duration
                       (get-syn-item (schema-reifier schema)))
                      (fix- *clock-tick* (schema-first-tick schema)))
                     (syn-item-format "after ~3D "
                                             (average-value
                                              (syn-item-on-duration
                                               (get-syn-item
                                                (schema-reifier schema)))))
                     (syn-item-format "first-tick set ~5D" *clock-tick*)
                     (setf (schema-first-tick schema) *clock-tick*))))
           (syn-item-format "syn not activated "))
       ;; not locally consistent
       (if (and (schema-activated-p schema)
                (flag-falsep (schema-result-empty schema))
                (schema-result-satisfied-p schema))
           (progn
             (syn-item-format "no syn update consistency before ~A "
                                     (schema-lc-consy-unparse schema))
             (schema-lc-consy-update
              schema (schema-succeeded-last-p schema))
             (syn-item-format "after ~A "
                                     (schema-lc-consy-unparse schema))
             (if (and (schema-lc-consy-high-p schema)
                      (weighted-rate-low-p (schema-reliability schema)))
                 (setf (schema-lcly-cons schema)
                       (make-flag-true)
                       (schema-reifier schema)
                       (make-syn-item schema-index))))
         (syn-item-format
          "no syn not activated or no non-empty satisfied result")))
     (syn-item-format "~%")
     (if (schema-activated-p schema)
         (setf (schema-succeeded-last schema)
               (schema-result-satisfied schema)))))
```

;;; output function: show–reliable–schemas

show-reliable-schemas

```
(defun show-reliable-schemas ()
  (main-format "reliable schemas:~%")
  (dotimes
   (x *schema-number*)
   (let ((schema (get-schema x)))
     (if (weighted-rate-high-p (schema-reliability schema))
         (main-format "~5D~%" x))))
  (main-format "~%"))
```

run

```
;;; main function: run

(defun run
  (how-many-times outputfilenameprefix randomstatefilename selector)

  (init-everything randomstatefilename selector)
  (dotimes
   (y how-many-times)
   (time
   (let ((filename
          (concatenate 'string
                       outputfilenameprefix
                       (format nil "~3,'0D" y)
                       ".out")))
     (with-open-file
      (out-file filename
                :direction :output
                :if-exists :supersede)
      (setf *output-stream* out-file)
      (dotimes
       (x 50)
       ;; even probability of actions
       (let ((action-index (random 10)))
         (format t "~D~%" (+ x (* y 50)))
         (show-items-format "initial state ~%")
         (show-items)
         (schema-update-applicable)
         (update-accessibility)
         (funcall (get-action action-index))
         (setf *clock-tick* (microworld:clock-tick))
         (item-update-state)
         (conj-update-state)
         (schema-update-activated action-index)
         (syn-item-update-state)
         ;;; for debugging
         (debug-format
          "~%~A~%"
          (schema-print-name (get-schema action-index)))
         (print-state-array *microworld-state* 25)
         (show-items)
         (schema-update-all-results)
         (schema-update-reliability)
         (update-predicted-results)
         (print-predicted-results)
         (schema-update-ext-item-stats)
         (schema-update-ext-conj-stats)
         (maybe-spinoff-schema)
         (maybe-make-conjs)
         (maybe-make-syn-items)))
     (show-reliable-schemas))))
   (format t "number of schemas:  ~D~%" *schema-number*)
   (format t "flushing output to file and closing...~%"))
  (setf *output-stream* t))
```

```
;;; microworld testing functions:
;;; test1-init-everything, test2-init-everything, test3-init-everything
```

test1-init-everything

```
(defun test1-init-everything ()
  (setf *clock-tick* (microworld:test1-init-world))
  (setf *conj-number* 0)
  (init-item)
  (init-schema-and-action)
  'test1-init-everything-finished)
```

test2-init-everything

```
(defun test2-init-everything ()
  (setf *clock-tick* (microworld:test2-init-world))
  (setf *conj-number* 0)
  (init-item)
  (init-schema-and-action)
  'test2-init-everything-finished)
```

test3-init-everything

```
(defun test3-init-everything ()
  (setf *clock-tick* (microworld:test3-init-world))
  (setf *conj-number* 0)
  (init-item)
  (init-schema-and-action)
  'test3-init-everything-finished)

;;; microworld testing function: schema-microworld-test
```

schema-microworld-test

```
(defun schema-microworld-test
  (&optional (screen-output nil))

  (with-open-file
   (out-file "mwtest.out"
             :direction :output
             :if-exists :supersede)
   (if screen-output
       (setf *output-stream* t)
     (setf *output-stream* out-file))
   (test1-init-everything)
   (item-update-state)
   (show-items-format "initial state ~%")
   (show-items)
```

```
;;; schema-microworld-test continued

  ;; 0 handf 1 handb 2 handr 3 handl
  ;; 4 eyef  5 eyeb  6 eyer  7 eyel
  ;; 8 grasp 9 ungrasp
  (dolist
   (elem
    (list

     ;; hand movement
     ;; test each corner and edge

     1         2         2
     1
     0         0         0
     2
     3         3         3
     0
     1         1         1
     3

     ;; hand movement
     ;; empty hand -- try to move into occupied space

     2         0                   ;r f
     2         0         3         ;r f l
     0         3         1         ;f l b
     3         1         2         ;l b r
     1         2                   ;b r

     ;; eye movement -- move through each corner testing all boundaries

     6         6         5
     5
     4         4         4
     6
     7         7         7
     4
     5         5         5
     7
     6         6         6
     ))
   (let ((schema (get-schema elem)))
     (funcall (schema-action schema))
     (item-update-state)
     ;;; for debugging
     (debug-format "~%~A~%" (schema-print-name schema))
     (show-items)))
  (test2-init-everything)
  (item-update-state)
  (show-items-format "initial state ~%")
  (show-items)
```

```
;;; schema-microworld-test continued

  ;; 0 handf 1 handb 2 handr 3 handl
  ;; 4 eyef  5 eyeb  6 eyer  7 eyel
  ;; 8 grasp 9 ungrasp
  (dolist
   (elem
    (list

     ;; object/hand movement
     ;; try to grab movable with closed hand and immovable object

     2         8         0
     8         0

     9         3
     8         2

     ;; now ungrasp hand and pick up movable object, then check
     ;; movement into currently occupied positions, each direction
     ;; both hand and object (except hand moving left into object,
     ;; and grasped object moving right into object)
     ;; tests continue in next do statement

     2         9         1
     8         3         0  ; move grasped object forward into object
     3         0            ; move hand forward into object

     2         2         0
     3                      ; move grasped object left into object
     ))
   (let ((schema (get-schema elem)))
     (funcall (schema-action schema))
     (item-update-state)
     ;;; for debugging
     (debug-format "~%~A~%" (schema-print-name schema))
     (show-items)))
  (test3-init-everything)
  (item-update-state)
  (show-items-format "initial state ~%")
  (show-items)
```

;;; schema-microworld-test continued

```
   ;; 0 handf 1 handb 2 handr 3 handl
   ;; 4 eyef  5 eyeb  6 eyer  7 eyel
   ;; 8 grasp 9 ungrasp
   (dolist
    (elem
     (list

      ;; continue movement checks into occupied positions with grasped
      ;; object

      8         1                ; move hand backward into object
      2         1                ; move grasped object backward into object

      3         3         1
      2                          ; move hand right into object
      ))
    (let ((schema (get-schema elem)))
      (funcall (schema-action schema))
      (item-update-state)
      ;;; for debugging
      (debug-format "~%~A~%" (schema-print-name schema))
      (show-items)))
  (setf *output-stream* t)))
```

A.3 Support for fixnum math

```
;;; fix.lisp
;;; developed by Robert Ramstad with help from
;;; Paul Anagnostopoulous, Digital Equipment Corporation

;;; this file allows for easy optimization of procedures using fixnum
;;; arithmetic

(in-package 'fix)

(export
 '(fix=
   fix/=
   fix>
   fix>=
   fix<
   fix<=
   fix+
   fix-
   fix*
   fix/
   fix1+
   fix1-
   fixplusp
   fixminusp
   fixrsh))
```

```
;;; use of these macros helps math performance without adding extra
;;; typing — while possibly redundant given the use of declarations
;;; throughout, using these macros insures that the compiler generates
;;; code which uses fixnum math and compare operations exclusively

(defmacro fix= (x y)                                                          fix=
  '(= (the fixnum ,x) (the fixnum ,y)))

(defmacro fix/= (x y)                                                         fix/=
  '(/= (the fixnum ,x) (the fixnum ,y)))

(defmacro fix> (x y)                                                          fix>
  '(> (the fixnum ,x) (the fixnum ,y)))

(defmacro fix>= (x y)                                                         fix>=
  '(>= (the fixnum ,x) (the fixnum ,y)))

(defmacro fix< (x y)                                                          fix<
  '(< (the fixnum ,x) (the fixnum ,y)))

(defmacro fix<= (x y)                                                         fix<=
  '(<= (the fixnum ,x) (the fixnum ,y)))

(defmacro fix+ (x y)                                                          fix+
  '(the fixnum (+ (the fixnum ,x) (the fixnum ,y))))

(defmacro fix- (x &rest more-nums)                                            fix-
  '(the fixnum (- (the fixnum ,x) ,@(mapcar #'(lambda (n) '(the fixnum ,n))
                                            more-nums))))
(defmacro fix* (x y)                                                          fix*
  '(the fixnum (* (the fixnum ,x) (the fixnum ,y))))

(defmacro fix/ (x y)                                                          fix/
  '(the fixnum (/ (the fixnum ,x) (the fixnum ,y))))

(defmacro fix1+ (x)                                                           fix1+
  '(the fixnum (1+ (the fixnum ,x))))

(defmacro fix1- (x)                                                           fix1-
  '(the fixnum (1- (the fixnum ,x))))

(defmacro fixplusp (x)                                                        fixplusp
  '(plusp (the fixnum ,x)))

(defmacro fixminusp (x)                                                       fixminusp
  '(minusp (the fixnum ,x)))

(defmacro fixrsh (x)                                                          fixrsh
  '(the fixnum (ash (the fixnum ,x) -1)))
```

A.4 Support for float math

```
;;; float.lisp
;;; developed by Robert Ramstad with help from
;;; Paul Anagnostopoulous, Digital Equipment Corporation

;;; this file allows for easy optimization of procedures using
;;; short–float arithmetic

(in–package 'float)

(export
 '(float=
   float/=
   float>
   float>=
   float<
   float<=
   float+
   float–
   float*
   float/
   float1+
   float1–))
```

```
;;; use of these macros helps math performance without adding extra
;;; typing - while possibly redundant given the use of declarations
;;; throughout, using these macros insures that the compiler generates
;;; code which uses short-float math and compare operations exclusively

(defmacro float= (x y)                                                  float=
  `(= (the short-float ,x) (the short-float ,y)))

(defmacro float/= (x y)                                                 float/=
  `(/= (the short-float ,x) (the short-float ,y)))

(defmacro float> (x y)                                                  float>
  `(> (the short-float ,x) (the short-float ,y)))

(defmacro float>= (x y)                                                 float>=
  `(>= (the short-float ,x) (the short-float ,y)))

(defmacro float< (x y)                                                  float<
  `(< (the short-float ,x) (the short-float ,y)))

(defmacro float<= (x y)                                                 float<=
  `(<= (the short-float ,x) (the short-float ,y)))

(defmacro float+ (x y)                                                  float+
  `(the short-float (+ (the short-float ,x) (the short-float ,y))))

(defmacro float- (x &rest more-nums)                                    float-
  `(the short-float (- (the short-float ,x)
                       ,@(mapcar #'(lambda (n) `(the short-float ,n))
                                 more-nums))))
(defmacro float* (x y)                                                  float*
  `(the short-float (* (the short-float ,x) (the short-float ,y))))

(defmacro float/ (x y)                                                  float/
  `(the short-float (/ (the short-float ,x) (the short-float ,y))))

(defmacro float1+ (x)                                                   float1+
  `(the short-float (1+ (the short-float ,x))))

(defmacro float1- (x)                                                   float1-
  `(the short-float (1- (the short-float ,x))))
```

A.5 Result analysis

```
;;; ANALYZE.LISP
;;; Analysis of test run output program
;;; Part of the reimplementation of the CM2 implementation
;;; as defined and explained in
;;; "Made-up Minds: A Constructivist Approach to Artificial Intelligence"
;;; by Gary L. Drescher, as published by MIT Press 1991
;;; originally published as a Ph.D. thesis September 1989
;;; Massachusetts Institute of Technology

;;; Part of the Computer Science Masters' thesis for
;;; Robert Ramstad, September 1992
;;; Massachusetts Institute of Technology
;;; Thesis Advisors: Professor Ronald Rivest
;;;   and Bruce Foster, Digital Equipment Corporation
;;; Copyright (c) 1992, Robert Ramstad, All Rights Reserved.
;;; The author hereby grants to MIT permission to reproduce and to
;;; distribute copies of this document in whole or in part.

;;; this file contains a powerful processor used to analyze the
;;; test runs from the thesis, plus a random routine used to help
;;; get various bits of data into LaTeX format

;;; the program works by getting each structure referenced in the test
;;; run into one of two arrays, *blah-array* for schemas, and
;;; *blah2-array* for everything else

;;; each member of *blah-array* is then categorized (or not) and these
;;; categories are the basis for the output from the program

(in-package 'schema)
```

doit

```
(defun doit ()
  (init)
  (analyze "0s01ana.out" "0s01res.out")
  (init)
  (analyze "0s02ana.out" "0s02res.out")
  (init)
  (analyze "0s03ana.out" "0s03res.out")
  (init)
  (analyze "0s04ana.out" "0s04res.out")
  (init)
  (analyze "0s05ana.out" "0s05res.out")
  (init)
  (analyze "0s06ana.out" "0s06res.out")
  (init)
  (analyze "0s07ana.out" "0s07res.out")
  (init)
  (analyze "0s08ana.out" "0s08res.out")
  (init)
  (analyze "0s09ana.out" "0s09res.out")
  (init)
  (analyze "0s10ana.out" "0s10res.out")
  (init)
  (analyze "0s11ana.out" "0s11res.out")
  (init)
  (analyze "0s12ana.out" "0s12res.out")
  (init)
  (analyze "0s13ana.out" "0s13res.out")
  (init)
  (analyze "0s14ana.out" "0s14res.out")
  (init)
  (analyze "0s15ana.out" "0s15res.out")
  (init)
  (analyze "0s16ana.out" "0s16res.out")
  (init)
  (analyze "0s17ana.out" "0s17res.out")
  (init)
  (analyze "0s18ana.out" "0s18res.out")
  (init)
  (analyze "0s19ana.out" "0s19res.out")
  (init)
  (analyze "0s20ana.out" "0s20res.out"))
```

blah-number

blah2-number

reliable-block

```
(defvar *blah-number* 0)
(defvar *blah2-number* 0)
(defvar *reliable-block* 0)

(proclaim '(fixnum *blah-number* *blah2-number* *reliable-block*))
```

init

```
(defun init ()
  (setf *blah-number* 0)
  (setf *blah2-number* 0)
  (setf *reliable-block* 0))
```

```
;;; blah datatype

;;; all the slots in this structure should be self explanatory except
;;; for rel-idx, which is a measure of how reliable the schema is
;;; rel-idx is calculated by dividing the number of times the schema
;;;   was over .9 reliability at the end of a block of 50 clock ticks
;;;   by the number of such blocks that have occurred since the schema
;;;   was first created
```

blah-array

get-blah

```
(defvar *blah-array* (make-array 5000))
(defmacro get-blah (x) `(aref *blah-array* ,x))
```

blah2-array

get-blah2

```
(defvar *blah2-array* (make-array 5000))
(defmacro get-blah2 (x) `(aref *blah2-array* ,x))
```

blah

```
(defstruct blah
  (creation-time  0 :type fixnum)
  (creation-block  0 :type fixnum)
  (print-name     "" :type string)
  (context       "" :type string)
  (action        "" :type string)
  (result        "" :type string)
  (category      99 :type fixnum)
  (reliable-blocks '() :type list)
  (rel-idx 0.0 :type short-float))
```

print-blah

```
(defun print-blah (blah &optional (out-stream t))
  (format out-stream "time:  ~5D  schema:  ~30A rel-idx:  ~5,3F~%"
          (blah-creation-time blah)
          (blah-print-name blah)
          (blah-rel-idx blah)))
```

print-blah2-short

```
(defun print-blah2-short (blah &optional (out-stream t))
  (format out-stream "time:  ~5D  schema:  ~30A~%"
          (blah-creation-time blah)
          (blah-print-name blah)))
```

print-list-of-fixnums

```
(defun print-list-of-fixnums
  (list &optional (out-stream t))

  (dolist
   (elem list)
   (format out-stream "~3D " elem))
  (format out-stream "~%"))
```

;;; blah datatype functions continued

make-and-stash-blah

```
(defun make-and-stash-blah (&optional category)
  (let ((currentblah (make-blah)))
    (setf (get-blah *blah-number*)
          currentblah
          *blah-number*
          (fix1+ *blah-number*))
    (if category
        (setf (blah-category currentblah) category))
    currentblah))
```

make-and-stash-blah2

```
(defun make-and-stash-blah2 (&optional category)
  (let ((currentblah (make-blah)))
    (setf (get-blah2 *blah2-number*)
          currentblah
          *blah2-number*
          (fix1+ *blah2-number*))
    (if category
        (setf (blah-category currentblah) category))
    currentblah))
```

```
;;; get-category-print-name returns a string for every category number
;;; used by the analysis program
```

get-category-print-name

```
(defun get-category-print-name (category)
  (cond ((fix= category 0) "initial schema")
        ((fix= category 1) "grasping schema")
        ((fix= category 2) "visual shift schema")
        ((fix= category 3) "visual shift limit schema")
        ((fix= category 4) "foveal shift schema")
        ((fix= category 5) "coarse to detailed visual shift schema")
        ((fix= category 6) "visual network schema")
        ((fix= category 7) "hand network schema")
        ((fix= category 8) "negative consequence schema")
        ((fix= category 9) "hand to body schema")
        ((fix= category 10)
         "coarse seeing hand motion schema")
        ((fix= category 11)
         "detailed seeing hand motion schema")
        ((fix= category 12)
         "seeing the body via coarse visual items")
        ((fix= category 13)
         "seeing the body via detailed visual items")
        ((fix= category 14)
         "using the body as a visual reference point")
        ((fix= category 15)
         "hand position required for a given hp translation")
        ((fix= category 16)
         "gaze position required for a given vp translation")
        ((fix= category 17)
         "coarse visual item required for a given vf translation")
        ((fix= category 18)
         "coarse visual item required for a given fov translation")
        ((fix= category 19)
         "detailed visual item required for a given vf translation")
        ((fix= category 20)
         "foveal region relationship with coarse visual field")
        ((fix= category 21)
         "breaking up visual region by where objects are seen")
        ((fix= category 22)
         "hand has to be in a given position to touch the body")
        ((fix= category 23)
         "hand cannot push an object out of the way")
        ((fix= category 24)
         "detailed visual item required for a given fov translation")
        ((fix= category 25)
         "hand in front of body and moving against it")
        ((fix= category 26)
         "hand movement relating coarse to detailed visual items")
        (t (error "undefined category code"))))
```

;;; main routine: analyze

analyze

```
(defun analyze (outfilename &rest infilenames)
  ;; initialize the initial schemas and categorize them
  (let ((handr-blah (make-and-stash-blah 0))
       (handl-blah (make-and-stash-blah 0))
       (handf-blah (make-and-stash-blah 0))
       (handb-blah (make-and-stash-blah 0))
       (eyer-blah (make-and-stash-blah 0))
       (eyel-blah (make-and-stash-blah 0))
       (eyef-blah (make-and-stash-blah 0))
       (eyeb-blah (make-and-stash-blah 0))
       (grasp-blah (make-and-stash-blah 0))
       (ungrasp-blah (make-and-stash-blah 0)))
   (setf (blah-action handr-blah) "handr"
         (blah-action handl-blah) "handl"
         (blah-action handf-blah) "handf"
         (blah-action handb-blah) "handb"
         (blah-action eyer-blah) "eyer"
         (blah-action eyel-blah) "eyel"
         (blah-action eyef-blah) "eyef"
         (blah-action eyeb-blah) "eyeb"
         (blah-action grasp-blah) "grasp"
         (blah-action ungrasp-blah) "ungrasp"
         (blah-print-name handr-blah) "/handr/"
         (blah-print-name handl-blah) "/handl/"
         (blah-print-name handf-blah) "/handf/"
         (blah-print-name handb-blah) "/handb/"
         (blah-print-name eyer-blah) "/eyer/"
         (blah-print-name eyel-blah) "/eyel/"
         (blah-print-name eyef-blah) "/eyef/"
         (blah-print-name eyeb-blah) "/eyeb/"
         (blah-print-name grasp-blah) "/grasp/"
         (blah-print-name ungrasp-blah) "/ungrasp/"
         (blah-creation-block handr-blah) 1
         (blah-creation-block handl-blah) 1
         (blah-creation-block handf-blah) 1
         (blah-creation-block handb-blah) 1
         (blah-creation-block eyer-blah) 1
         (blah-creation-block eyel-blah) 1
         (blah-creation-block eyef-blah) 1
         (blah-creation-block eyeb-blah) 1
         (blah-creation-block grasp-blah) 1
         (blah-creation-block ungrasp-blah) 1))
 ;; open the output file
 (with-open-file
  (out-file outfilename
            :direction :output
            :if-exists :supersede)
```

```
;;; analyze routine continued

   ;; process the input files in succession
   (dolist
    (infile infilenames)
    (with-open-file
     (in-file infile
              :direction :input)
     ;; process each input file, creating blah structures for each
     ;; schema encountered, and blah2 structures for everything else
     ;; when reliability info is encountered, append it to the
     ;; appropriate blah structure for later rel-idx calculation
     (do ((time -1)
          (line nil)
          (status nil))
         ((progn
            (setf time (read in-file nil nil))
            (multiple-value-setq (line status) (read-line in-file nil nil))
            (if line status t))
          "Thanks for using analyze, hope it was helpful!")
         (cond ((eq time 'reliable)
                (setf *reliable-block* (fix1+ *reliable-block*))
                (format t "finished through time ~5D~%"
                        (fix* 50 *reliable-block*))
                (do ((schema-number-string nil (read-line in-file nil nil)))
                    ((string= "" schema-number-string) "finished")
                    (if schema-number-string
                        (let ((schema-number
                               (read-from-string schema-number-string)))
                          (setf (blah-reliable-blocks
                                 (get-blah schema-number))
                                (append
                                 (blah-reliable-blocks
                                  (get-blah schema-number))
                                 (list *reliable-block*)))))))
               ((string= "composite-" (subseq line 0 10))
                (let ((new-blah (make-and-stash-blah2)))
                  (setf (blah-creation-time new-blah) time
                        (blah-creation-block new-blah) *reliable-block*
                        (blah-category new-blah) 0
                        (blah-print-name new-blah)
                        (concatenate 'string "/<" (line-schema line) ">/"))))
               ((string= "syn item " (subseq line 0 9))
                (let ((new-blah (make-and-stash-blah2)))
                  (setf (blah-creation-time new-blah) time
                        (blah-creation-block new-blah) *reliable-block*
                        (blah-category new-blah) 1
                        (blah-print-name new-blah) (line-schema line))))
               (t
                (let ((new-blah (make-and-stash-blah)))
                  (setf (blah-creation-time new-blah) time
                        (blah-creation-block new-blah) *reliable-block*
                        (blah-print-name new-blah) (line-schema line))))))))
```

```
;;; analyze routine continued

   ;; blah-update moves the components of print-name into the context,
   ;; action and result slots, and also calculates the rel-idx
   (dotimes
    (x *blah-number*)
    (let ((blah (get-blah x)))
      (blah-update blah)))

   (dotimes
    (x *blah2-number*)
    (let ((blah (get-blah2 x)))
      (blah-update blah)))

   ;; categorize all the elements in the *blah-array*
   (dotimes
    (x *blah-number*)
    (let* ((blah (get-blah x))
           (context (blah-context blah))
           (action (blah-action blah))
           (result (blah-result blah))
           (print-name (blah-print-name blah)))
      (declare (string context action result))
      (let ((context-len (length context))
            (action-len (length action))
            (result-len (length result))
            (print-len (length print-name)))
        (declare (fixnum context-len action-len result-len))
```

```
;;; analyze routine continued

        ;; if not already categorized
        (if (fix= 99 (blah-category blah))
            ;; place in one of the following categories
            (cond
             ;; any schema we don't want to see, set it's category to 98
             ((or (string= (subseq print-name 0 1) "[")
                  (string= (subseq print-name 0 2) "-[")
                  (string=
                   (subseq print-name (fix1- print-len) print-len)
                   "]"))
              (setf (blah-category blah) 98))
             ;; vf??/eye?/vf?? same items
             ((and (fix= 4 context-len)
                   (fix= 4 action-len)
                   (fix= 4 result-len)
                   (string= "vf" (subseq context 0 2))
                   (string= "eye" (subseq action 0 3))
                   (string= "vf" (subseq result 0 2))
                   (string= context result))
              (setf (blah-category blah) 98))
             ;; -vf??/eye?/-vf?? same items
             ((and (fix= 5 context-len)
                   (fix= 4 action-len)
                   (fix= 5 result-len)
                   (string= "-vf" (subseq context 0 3))
                   (string= "eye" (subseq action 0 3))
                   (string= "-vf" (subseq result 0 3))
                   (string= context result))
              (setf (blah-category blah) 98))
             ;; fov???/eye?/fov??? and same fov letter
             ((and (fix= 6 context-len)
                   (fix= 4 action-len)
                   (fix= 6 result-len)
                   (string= "fov" (subseq context 0 3))
                   (string= "eye" (subseq action 0 3))
                   (string= "fov" (subseq result 0 3))
                   (string= (subseq context 0 4) (subseq result 0 4)))
              (setf (blah-category blah) 98))
             ;; -hp??/hand?/hp??
             ((and (fix= 5 context-len)
                   (fix= 5 action-len)
                   (fix= 4 result-len)
                   (string= "-hp" (subseq context 0 3))
                   (string= "hand" (subseq action 0 4))
                   (string= "hp" (subseq result 0 2)))
              (setf (blah-category blah) 98))
```

```
;;; analyze routine categorization continued

              ;; grasping schemas
              ;; */grasp/*, */ungrasp/*
              ((or (string= "grasp" action)
                   (string= "ungrasp" action))
               (setf (blah-category blah) 1))

              ;; visual shift schemas
              ;; vf??/eye?/vf??
              ;; two items adjacent based on eye movement
              ((and (fix= 4 context-len)
                    (fix= 4 action-len)
                    (fix= 4 result-len)
                    (string= "vf" (subseq context 0 2))
                    (string= "eye" (subseq action 0 3))
                    (string= "vf" (subseq result 0 2))
                    (relationship-vf-eye-vf (subseq context 2 4)
                                            (subseq action 3 4)
                                            (subseq result 2 4)))
               (setf (blah-category blah) 2))
```

```
;;; analyze routine categorization continued

                ;; visual shift limit schemas
                ;; vf??&-vp??/eye?/vf??
                ;; two items adjacent based on eye movement
                ;; and -vp is valid maximum for the eye movement
                ;; first digit 0 for l, 2 for r
                ;; second digit 0 for b, 2 for f
                ((and (fix= 10 context-len)
                      (fix= 4 action-len)
                      (fix= 4 result-len)
                      (string= "eye" (subseq action 0 3))
                      (string= "vf" (subseq result 0 2))
                      (or (and (string= "vf" (subseq context 0 2))
                               (string= "&-vp" (subseq context 4 8))
                               (relationship-vf-eye-vf
                                (subseq context 2 4)
                                (subseq action 3 4)
                                (subseq result 2 4))
                               (or (and (string= (subseq action 3 4) "b")
                                        (string= (subseq context 9 10) "0"))
                                   (and (string= (subseq action 3 4) "f")
                                        (string= (subseq context 9 10) "2"))
                                   (and (string= (subseq action 3 4) "l")
                                        (string= (subseq context 8 9) "0"))
                                   (and (string= (subseq action 3 4) "r")
                                        (string= (subseq context 8 9) "2"))))
                          (and (string= "-vp" (subseq context 0 3))
                               (string= "&vf" (subseq context 5 8))
                               (relationship-vf-eye-vf
                                (subseq context 8 10)
                                (subseq action 3 4)
                                (subseq result 2 4))
                               (or (and (string= (subseq action 3 4) "b")
                                        (string= (subseq context 4 5) "0"))
                                   (and (string= (subseq action 3 4) "f")
                                        (string= (subseq context 4 5) "2"))
                                   (and (string= (subseq action 3 4) "l")
                                        (string= (subseq context 3 4) "0"))
                                   (and (string= (subseq action 3 4) "r")
                                        (string= (subseq context 3 4) "2"))))))
                 (setf (blah-category blah) 3))
```

```
;;; analyze routine categorization continued

                        ;; foveal shift schemas
                        ;; fov???*/eye?/fov???
                        ;; two foveal letters adjacent
                        ((and (fix< 5 context-len)
                              (fix= 4 action-len)
                              (fix= 6 result-len)
                              (string= "fov" (subseq context 0 3))
                              (string= "eye" (subseq action 0 3))
                              (string= "fov" (subseq result 0 3))
                              (relationship-fov-eye-fov
                               (subseq context 3 4)
                               (subseq action 3 4)
                               (subseq result 3 4)))
                         (setf (blah-category blah) 4))
```

;;; analyze routine categorization continued

```
;; detail shift schemas
;; vf??/eye?/fov???
;; correct relationship between coarse and detailed
;; vf??&fov???/eye?/fov???
;; two foveal letters adjacent
((and (fix= 4 context-len)
      (fix= 4 action-len)
      (fix= 6 result-len)
      (string= "vf" (subseq context 0 2))
      (string= "eye" (subseq action 0 3))
      (string= "fov" (subseq result 0 3))
      (relationship-vf-eye-fov (subseq context 2 4)
                               (subseq action 3 4)
                               (subseq result 3 4)))
 (setf (blah-category blah) 5))
((and (fix= 11 context-len)
      (fix= 4 action-len)
      (fix= 6 result-len)
      (string= "eye" (subseq action 0 3))
      (string= "fov" (subseq result 0 3))
      (or (and (string= "vf" (subseq context 0 2))
               (string= "&fov" (subseq context 4 8))
               (relationship-vf-eye-fov
                (subseq context 2 4)
                (subseq action 3 4)
                (subseq result 3 4))
               (relationship-fov-eye-fov
                (subseq context 8 9)
                (subseq action 3 4)
                (subseq result 3 4)))
          (and (string= "fov" (subseq context 0 3))
               (string= "&vf" (subseq context 5 8))
               (relationship-vf-eye-fov
                (subseq context 8 10)
                (subseq action 3 4)
                (subseq result 3 4))
               (relationship-fov-eye-fov
                (subseq context 3 4)
                (subseq action 3 4)
                (subseq result 3 4)))))
 (setf (blah-category blah) 5))
```

;;; analyze routine categorization continued

```
;; visual network schemas
;; vp??/eye?/vp??
;; two items adjacent or identical (gaze limit)
;; vf??/eye?/vp?? or
;; vf??&vp??/eye?/vp??
;; two items adjacent or identical (gaze limit)
;; vf is due to body relationship
((and (fix= 4 context-len)
      (fix= 4 action-len)
      (fix= 4 result-len)
      (string= "vp" (subseq context 0 2))
      (string= "eye" (subseq action 0 3))
      (string= "vp" (subseq result 0 2))
      (relationship-vp-eye-vp (subseq context 2 4)
                              (subseq action 3 4)
                              (subseq result 2 4)))
 (setf (blah-category blah) 6))
((and (fix= 4 action-len)
      (fix= 4 result-len)
      (string= "eye" (subseq action 0 3))
      (string= "vp" (subseq result 0 2))
      (or (and (fix= 4 context-len)
               (string= "vf" (subseq context 0 2))
               (relationship-vf-eye-vp
                (subseq context 2 4)
                (subseq action 3 4)
                (subseq result 2 4)))
          (and (fix= 9 context-len)
               (string= "vf" (subseq context 0 2))
               (string= "&vp" (subseq context 4 7))
               (relationship-vf-eye-vp
                (subseq context 2 4)
                (subseq action 3 4)
                (subseq result 2 4))
               (relationship-vp-eye-vp
                (subseq context 7 9)
                (subseq action 3 4)
                (subseq result 2 4)))
          (and (fix= 9 context-len)
               (string= "vp" (subseq context 0 2))
               (string= "&vf" (subseq context 4 7))
               (relationship-vf-eye-vp
                (subseq context 7 9)
                (subseq action 3 4)
                (subseq result 2 4))
               (relationship-vp-eye-vp
                (subseq context 2 4)
                (subseq action 3 4)
                (subseq result 2 4)))))
(setf (blah-category blah) 6))
```

```
;; hand network schemas
;; hp??/hand?/hp??
;; two items adjacent or identical
;; (taste? or tactb or bodyf)/hand?/hp??
((and (fix= 4 context-len)
      (fix= 5 action-len)
      (fix= 4 result-len)
      (string= "hp" (subseq context 0 2))
      (string= "hand" (subseq action 0 4))
      (string= "hp" (subseq result 0 2))
      (relationship-hp-hand-hp
       (subseq context 2 4)
       (subseq action 4 5)
       (subseq result 2 4)))
 (setf (blah-category blah) 7))
((and (fix= 5 action-len)
      (fix= 4 result-len)
      (string= "hand" (subseq action 0 4))
      (string= "hp" (subseq result 0 2))
      (or (and (fix= 6 context-len)
               (string= "taste" (subseq context 0 5)))
          (string= "tactb" context)
          (string= "bodyf" context))
      (relationship-body-hand-hp
       (subseq action 4 5)
       (subseq result 2 4)))
 (setf (blah-category blah) 7))

;; negative consequence schemas
;; x/eye?/-x or x/hand?/-x
((and (string= result (concatenate 'string "-" context))
      (or (and (fix= 4 action-len)
               (string= "eye" (subseq action 0 3)))
          (and (fix= 5 action-len)
               (string= "hand" (subseq action 0 4)))))
 (setf (blah-category blah) 8))

;; hand to body schemas
;; hp??/hand?/(taste? or tactb or bodyf)
((and (fix= 4 context-len)
      (fix= 5 action-len)
      (string= "hp" (subseq context 0 2))
      (string= "hand" (subseq action 0 4))
      (relationship-hp-hand-body
       (subseq context 2 4)
       (subseq action 4 5))
      (or (and (fix= 6 result-len)
               (string= "taste" (subseq result 0 5)))
          (string= "tactb" result)
          (string= "bodyf" result)))
 (setf (blah-category blah) 9))
```

```
;;; analyze routine categorization continued

             ;; coarse seeing hand motion schemas
             ;; vf??/hand?/vf??
             ;; the two items relate to one another correctly
             ((and (fix= 4 context-len)
                   (fix= 5 action-len)
                   (fix= 4 result-len)
                   (string= "vf" (subseq context 0 2))
                   (string= "hand" (subseq action 0 4))
                   (string= "vf" (subseq result 0 2))
                   (relationship-vf-hand-vf
                    (subseq context 2 4)
                    (subseq action 4 5)
                    (subseq result 2 4)))
              (setf (blah-category blah) 10))

             ;; detailed seeing hand motion schemas
             ;; fov???*/hand?/fov???*
             ;; foveal regions related correctly
             ((and (fix< 5 context-len)
                   (fix= 5 action-len)
                   (fix< 5 result-len)
                   (string= "fov" (subseq context 0 3))
                   (string= "hand" (subseq action 0 4))
                   (string= "fov" (subseq result 0 3))
                   (relationship-fov-hand-fov
                    (subseq context 3 4)
                    (subseq action 4 5)
                    (subseq result 3 4)))
              (setf (blah-category blah) 11))

             ;; seeing the body coarse
             ;; vp??/eye?/vf??
             ;; each correct relative to the body position
             ((and (fix= 4 context-len)
                   (fix= 4 action-len)
                   (fix= 4 result-len)
                   (string= "vp" (subseq context 0 2))
                   (string= "eye" (subseq action 0 3))
                   (string= "vf" (subseq result 0 2))
                   (relationship-vp-eye-vf
                    (subseq context 2 4)
                    (subseq action 3 4)
                    (subseq result 2 4)))
              (setf (blah-category blah) 12))
```

```
;;; analyze routine categorization continued

            ;; seeing the body detailed
            ;; vp??/eye?/fov???
            ;; each correct relative to the body position
            ((and (fix= 4 context-len)
                  (fix= 4 action-len)
                  (fix= 6 result-len)
                  (string= "vp" (subseq context 0 2))
                  (string= "eye" (subseq action 0 3))
                  (string= "fov" (subseq result 0 3))
                  (relationship-vp-eye-fov
                   (subseq context 2 4)
                   (subseq action 3 4)
                   (subseq result 3 4)))
             (setf (blah-category blah) 13))

            ;; body as visual reference point
            ;; -fov???/eye?/-vp?? or
            ;; fov???/eye?/vp??
            ((and (fix= 7 context-len)
                  (fix= 4 action-len)
                  (fix= 5 result-len)
                  (string= "-fov" (subseq context 0 4))
                  (string= "eye" (subseq action 0 3))
                  (string= "-vp" (subseq result 0 3))
                  (relationship-fov-eye-vp
                   (subseq context 4 5)
                   (subseq action 3 4)
                   (subseq result 3 5)))
             (setf (blah-category blah) 14))
            ((and (fix= 6 context-len)
                  (fix= 4 action-len)
                  (fix= 4 result-len)
                  (string= "fov" (subseq context 0 3))
                  (string= "eye" (subseq action 0 3))
                  (string= "vp" (subseq result 0 2))
                  (relationship-fov-eye-vp
                   (subseq context 3 4)
                   (subseq action 3 4)
                   (subseq result 2 4)))
             (setf (blah-category blah) 14))
```

```
;;; analyze routine categorization continued

             ;; hand position required for hp transition
             ;; -hp??/hand?/-hp??
             ;; items adjacent and not identical
             ;; (-taste? or -tactb or -bodyf)/hand?/-hp??
             ((and (fix= 5 context-len)
                   (fix= 5 action-len)
                   (fix= 5 result-len)
                   (string= "-hp" (subseq context 0 3))
                   (string= "hand" (subseq action 0 4))
                   (string= "-hp" (subseq result 0 3))
                   (relationship-hp-hand-hp
                    (subseq context 3 5)
                    (subseq action 4 5)
                    (subseq result 3 5)))
              (setf (blah-category blah) 15))
             ((and (fix= 5 action-len)
                   (fix= 5 result-len)
                   (string= "hand" (subseq action 0 4))
                   (string= "-hp" (subseq result 0 3))
                   (or (and (fix= 7 context-len)
                            (string= "-taste" (subseq context 0 6)))
                       (string= "-tactb" context)
                       (string= "-bodyf" context))
                   (relationship-body-hand-hp
                    (subseq action 4 5)
                    (subseq result 3 5)))
              (setf (blah-category blah) 15))

             ;; gaze position required for vp transition
             ;; -vp??/eye?/-vp??
             ((and (fix= 5 context-len)
                   (fix= 4 action-len)
                   (fix= 5 result-len)
                   (string= "-vp" (subseq context 0 3))
                   (string= "eye" (subseq action 0 3))
                   (string= "-vp" (subseq result 0 3))
                   (relationship-vp-eye-vp
                    (subseq context 3 5)
                    (subseq action 3 4)
                    (subseq result 3 5)))
              (setf (blah-category blah) 16))
```

```
;;; analyze routine categorization continued

          ;; coarse visual item required for vf transition
          ;; -vf??/eye?/-vf??
          ((and (fix= 5 context-len)
                (fix= 4 action-len)
                (fix= 5 result-len)
                (string= "-vf" (subseq context 0 3))
                (string= "eye" (subseq action 0 3))
                (string= "-vf" (subseq result 0 3))
                (relationship-vf-eye-vf
                 (subseq context 3 5)
                 (subseq action 3 4)
                 (subseq result 3 5)))
           (setf (blah-category blah) 17))

          ;; coarse visual item required for fov transition
          ;; -vf??/eye?/-fov???
          ((and (fix= 5 context-len)
                (fix= 4 action-len)
                (fix= 7 result-len)
                (string= "-vf" (subseq context 0 3))
                (string= "eye" (subseq action 0 3))
                (string= "-fov" (subseq result 0 4))
                (relationship-vf-eye-fov
                 (subseq context 3 5)
                 (subseq action 3 4)
                 (subseq result 4 5)))
           (setf (blah-category blah) 18))

          ;; detailed visual item required for vf transition
          ;; -fov???/eye?/-vf???
          ((and (fix= 7 context-len)
                (fix= 4 action-len)
                (fix= 5 result-len)
                (string= "-fov" (subseq context 0 4))
                (string= "eye" (subseq action 0 3))
                (string= "-vf" (subseq result 0 3))
                (relationship-fov-eye-vf
                 (subseq context 4 5)
                 (subseq action 3 4)
                 (subseq result 3 5)))
           (setf (blah-category blah) 19))
```

```
;;; analyze routine categorization continued

            ;; relationship between foveal regions and coarse visual field
            ;; fov???/eye?/vf??
            ((and (fix= 6 context-len)
                  (fix= 4 action-len)
                  (fix= 4 result-len)
                  (string= "fov" (subseq context 0 3))
                  (string= "eye" (subseq action 0 3))
                  (string= "vf" (subseq result 0 2))
                  (relationship-fov-eye-vf
                   (subseq context 3 4)
                   (subseq action 3 4)
                   (subseq result 2 4)))
             (setf (blah-category blah) 20))

            ;; breaking up the visual region by where objects are seen
            ;; vp/eye?/vf or fov
            ;; -vp??/eye?/(-vf?? or -fov???)
            ((and (fix= 4 context-len)
                  (fix= 4 action-len)
                  (fix= 4 result-len)
                  (string= "vp" (subseq context 0 2))
                  (string= "eye" (subseq action 0 3))
                  (string= "vf" (subseq result 0 2)))
             (setf (blah-category blah) 21))
            ((and (fix= 4 context-len)
                  (fix= 4 action-len)
                  (fix= 6 result-len)
                  (string= "vp" (subseq context 0 2))
                  (string= "eye" (subseq action 0 3))
                  (string= "fov" (subseq result 0 3)))
             (setf (blah-category blah) 21))
            ((and (fix= 5 context-len)
                  (fix= 4 action-len)
                  (string= "-vp" (subseq context 0 3))
                  (string= "eye" (subseq action 0 3))
                  (or (and (fix= 5 result-len)
                           (string= "-vf" (subseq result 0 3)))
                      (and (fix= 7 result-len)
                           (string= "-fov" (subseq result 0 4)))))
             (setf (blah-category blah) 21))
```

;;; analyze routine categorization continued

```
;; hand has to be in a given position to touch the body
;; -hp??/hand?/(-taste? or -tactb or -bodyf)
((and (fix= 5 context-len)
      (fix= 5 action-len)
      (string= "-hp" (subseq context 0 3))
      (string= "hand" (subseq action 0 4))
      (relationship-hp-hand-body
       (subseq context 3 5)
       (subseq action 4 5))
      (or (and (fix= 7 result-len)
               (string= "-taste" (subseq result 0 6)))
          (and (fix= 6 result-len)
               (or (string= "-tactb" (subseq result 0 6))
                   (string= "-bodyf" (subseq result 0 6))))))
 (setf (blah-category blah) 22))

;; hand cannot push an object out of the way
;; or, if holding it, moves it along with it
;; tact?/hand?/tact? and ? is the same or
;; tactl/handl/text? or text?/handl/tactl
((and (fix= 5 context-len)
      (fix= 5 action-len)
      (fix= 5 result-len)
      (or (and (string= context result)
               (string= "tact" (subseq context 0 4))
               (string= "hand" (subseq action 0 4))
               (string= (subseq context 4 5)
                        (subseq action 4 5)))
          (and (string= context "tactl")
               (string= action "handl")
               (string= "text" (subseq result 0 4)))
          (and (string= "text" (subseq context 0 4))
               (string= action "handl")
               (string= result "tactl"))))
 (setf (blah-category blah) 23))
```

```
;;; analyze routine categorization continued

                        ;; required fov item for fov result schemas
                        ;; -fov???/eye?/-fov???
                        ;; two foveal letters adjacent
                        ((and (fix= 7 context-len)
                              (fix= 4 action-len)
                              (fix= 7 result-len)
                              (string= "-fov" (subseq context 0 4))
                              (string= "eye" (subseq action 0 3))
                              (string= "-fov" (subseq result 0 4))
                              (relationship-fov-eye-fov
                               (subseq context 4 5)
                               (subseq action 3 4)
                               (subseq result 4 5)))
                         (setf (blah-category blah) 24))

                        ;; hand in front of body and moving against it
                        ;; (tactb or bodyf or taste?)/handb/(tactb or bodyf or taste?)
                        ((and (fix< 4 context-len)
                              (fix= 5 action-len)
                              (fix< 4 result-len)
                              (string= "handb" action)
                              (or (string= "tactb" context)
                                  (string= "bodyf" context)
                                  (string= "taste" (subseq context 0 5)))
                              (or (string= "tactb" result)
                                  (string= "bodyf" result)
                                  (string= "taste" (subseq result 0 5))))
                         (setf (blah-category blah) 25))
```

```
;;; analyze routine categorization continued

             ;; coarse hand moves shows detail
             ;; vf??/hand?/fov???
             ;; detailed hand moves shows coarse
             ;; fov???/hand?/vf??
             ;; items related correctly
             ((and (fix= 5 action-len)
                   (string= "hand" (subseq action 0 4))
                   (or (and (fix= 4 context-len)
                            (string= "vf" (subseq context 0 2))
                            (fix= 6 result-len)
                            (string= "fov" (subseq result 0 3))
                            (relationship-vf-hand-fov
                             (subseq context 2 4)
                             (subseq action 4 5)
                             (subseq result 3 4)))
                       (and (fix= 6 context-len)
                            (string= "fov" (subseq context 0 3))
                            (fix= 4 result-len)
                            (string= "vf" (subseq result 0 2))
                            (relationship-fov-hand-vf
                             (subseq context 3 4)
                             (subseq action 4 5)
                             (subseq result 2 4)))))
              (setf (blah-category blah) 26))

             (t nil)))))
```

```
;;; analyze routine continued

   ;; finished categorizing, so do output
   ;; start by outputting a statistical summary for the test run
   (dotimes
    (category 27)
    (let ((elems 0)
          (rel-elems 0))
      (declare (fixnum elems rel-elems))
      (dotimes
       (x *blah-number*)
       (let ((blah (get-blah x)))
         (if (fix= (blah-category blah) category)
             (if (= (blah-rel-idx blah) 0.0)
                 (setf elems (fix1+ elems))
               (setf rel-elems (fix1+ rel-elems)
                     elems (fix1+ elems))))))
      (format out-file "category ~2D: ~60A~%"
              category (get-category-print-name category))
      (format out-file "             schemas:  ~3D reliable schemas:  ~3D~%"
              elems rel-elems)))
   (format out-file "~%~%")

   ;; then output the schemas belonging to each category
   (dotimes
    (category 27)
    (format out-file "category ~2D: ~60A~%"
            category (get-category-print-name category))
    (dotimes
     (x *blah-number*)
     (let ((blah (get-blah x)))
       (if (fix= category (blah-category blah))
           (print-blah blah out-file))))
    (format out-file "~%"))

   ;; finally, output other interesting schemas
   (format out-file
           "*** non-categorized non-empty context rel-idx above .7 ***~%")
   (dotimes
    (x *blah-number*)
    (let ((blah (get-blah x)))
      (if (and (fix= 99 (blah-category blah))
               (> (blah-rel-idx blah) .7)
               (fix/= 0 (length (blah-context blah))))
          (print-blah blah out-file))))
   (format out-file "~%")
   ))
```

line-schema

```
(defun line-schema (line)
  (let ((toggle nil)
        (end -1))
    (do* ((x        (fix1- (length line))   (fix1- x))
          (currchar (elt line x)            (elt line x)))
         ((and toggle (char= #\Space currchar)) (subseq line (fix1+ x) end))
         (cond ((and (not toggle)
                     (not (char= #\Space currchar)))
                (setf end (fix1+ x)
                      toggle t))
               (t nil)))))
```

blah-update

```
(defun blah-update (blah)
  (let ((toggle nil)
        (start 0)
        (line (blah-print-name blah)))
    (declare (string line))
    (setf line (delete #\> line :test #'char=))
    (setf line (delete #\< line :test #'char=))
    (setf line (delete #\] line :test #'char=))
    (setf line (delete #\[ line :test #'char=))
    (dotimes
     (x (length line))
     (let ((currentchar (elt line x)))
       (cond ((char= #\/ currentchar)
              (let ((str (subseq line start x)))
                (setf start (fix1+ x))
                (if toggle
                    (setf (blah-action blah)
                          str
                          (blah-result blah)
                          (subseq line (fix1+ x) (length line))
                          toggle
                          nil)
                  (setf (blah-context blah) str
                        toggle t)))))))
    (setf (blah-rel-idx blah)
          (float (/ (length (blah-reliable-blocks blah))
                    (fix1+ (fix- *reliable-block*
                                 (blah-creation-block blah))))))))
```

```
;;; functions used by analyze to express microworld relationships continued
;;; these should be easy to understand if the use of them is examined
;;;   and the examiner has a good grasp of microworld mechanics

;; relate coarse visual movement with gaze actions
```

relationship-vf-eye-vf

```
(defun relationship-vf-eye-vf (vf1 eye vf2)
  (let ((x1 (read-from-string (subseq vf1 0 1)))
        (y1 (read-from-string (subseq vf1 1 2)))
        (x2 (read-from-string (subseq vf2 0 1)))
        (y2 (read-from-string (subseq vf2 1 2))))
    (declare (fixnum x1 y1 x2 y2))
    (or (and (string= eye "l")
             (fix= (fix1+ x1) x2)
             (fix= y1 y2))
        (and (string= eye "r")
             (fix= x1 (fix1+ x2))
             (fix= y1 y2))
        (and (string= eye "b")
             (fix= x1 x2)
             (fix= (fix1+ y1) y2))
        (and (string= eye "f")
             (fix= x1 x2)
             (fix= y1 (fix1+ y2))))))

;; relate coarse visual movement with hand actions
```

relationship-vf-hand-vf

```
(defun relationship-vf-hand-vf (vf1 hand vf2)
  (let ((x1 (read-from-string (subseq vf1 0 1)))
        (y1 (read-from-string (subseq vf1 1 2)))
        (x2 (read-from-string (subseq vf2 0 1)))
        (y2 (read-from-string (subseq vf2 1 2))))
    (declare (fixnum x1 y1 x2 y2))
    (or (and (string= hand "l")
             (fix= x1 (fix1+ x2))
             (fix= y1 y2))
        (and (string= hand "r")
             (fix= (fix1+ x1) x2)
             (fix= y1 y2))
        (and (string= hand "b")
             (fix= x1 x2)
             (fix= y1 (fix1+ y2)))
        (and (string= hand "f")
             (fix= x1 x2)
             (fix= (fix1+ y1) y2)))))
```

```
;;; functions used by analyze to express microworld relationships continued

;; relate movement among foveal regions with gaze shift
;; two foveal regions are adjacent if one is "x" and the other is
;; any of "f" "b" "r" or "l"
(defun relationship-fov-eye-fov (fov1 eye fov2)       relationship-fov-eye-fov
  (and (not (string= fov1 fov2))
       (or (and (string= fov1 "x")
                (or (and (string= eye "l")
                         (string= fov2 "r"))
                    (and (string= eye "r")
                         (string= fov2 "l"))
                    (and (string= eye "b")
                         (string= fov2 "f"))
                    (and (string= eye "f")
                         (string= fov2 "b"))))
           (and (string= fov2 "x")
                (or (and (string= eye "l")
                         (string= fov1 "l"))
                    (and (string= eye "r")
                         (string= fov1 "r"))
                    (and (string= eye "b")
                         (string= fov1 "b"))
                    (and (string= eye "f")
                         (string= fov1 "f")))))))

;; relate movement among foveal regions with hand shift
(defun relationship-fov-hand-fov (fov1 hand fov2) relationship-fov-hand-fov
  (and (not (string= fov1 fov2))
       (or (and (string= fov1 "x")
                (or (and (string= hand "l")
                         (string= fov2 "l"))
                    (and (string= hand "r")
                         (string= fov2 "r"))
                    (and (string= hand "b")
                         (string= fov2 "b"))
                    (and (string= hand "f")
                         (string= fov2 "f"))))
           (and (string= fov2 "x")
                (or (and (string= hand "l")
                         (string= fov1 "r"))
                    (and (string= hand "r")
                         (string= fov1 "l"))
                    (and (string= hand "b")
                         (string= fov1 "f"))
                    (and (string= hand "f")
                         (string= fov1 "b")))))))
```

```
;;; functions used by analyze to express microworld relationships continued

;; relate coarse visual field to detailed visual field
```

relationship-vf-eye-fov

```
(defun relationship-vf-eye-fov (vf eye fov)
  (let ((xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2))))
    (declare (fixnum xvf yvf))
    (cond ((string= eye "l") (setf xvf (fix1+ xvf)))
          ((string= eye "r") (setf xvf (fix1- xvf)))
          ((string= eye "b") (setf yvf (fix1+ yvf)))
          ((string= eye "f") (setf yvf (fix1- yvf)))
          (t nil))
    (or (and (string= fov "x")
             (fix= xvf 2)
             (fix= yvf 2))
        (and (string= fov "l")
             (fix= xvf 1)
             (fix= yvf 2))
        (and (string= fov "r")
             (fix= xvf 3)
             (fix= yvf 2))
        (and (string= fov "b")
             (fix= xvf 2)
             (fix= yvf 1))
        (and (string= fov "f")
             (fix= xvf 2)
             (fix= yvf 3)))))

;; relate detailed visual field to coarse visual field
```

relationship-fov-eye-vf

```
(defun relationship-fov-eye-vf (fov eye vf)
  (let ((xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2))))
    (declare (fixnum xvf yvf))
    (cond ((string= eye "l") (setf xvf (fix1- xvf)))
          ((string= eye "r") (setf xvf (fix1+ xvf)))
          ((string= eye "b") (setf yvf (fix1- yvf)))
          ((string= eye "f") (setf yvf (fix1+ yvf)))
          (t nil))
    (or (and (string= fov "x")
             (fix= xvf 2)
             (fix= yvf 2))
        (and (string= fov "l")
             (fix= xvf 1)
             (fix= yvf 2))
        (and (string= fov "r")
             (fix= xvf 3)
             (fix= yvf 2))
        (and (string= fov "b")
             (fix= xvf 2)
             (fix= yvf 1))
        (and (string= fov "f")
             (fix= xvf 2)
             (fix= yvf 3)))))
```

```
;;; functions used by analyze to express microworld relationships continued

;; relate coarse visual field to detailed visual field via hand movement
(defun relationship-vf-hand-fov (vf hand fov)
  (let ((xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2))))
    (declare (fixnum xvf yvf))
    (cond ((string= hand "l") (setf xvf (fix1- xvf)))
          ((string= hand "r") (setf xvf (fix1+ xvf)))
          ((string= hand "b") (setf yvf (fix1- yvf)))
          ((string= hand "f") (setf yvf (fix1+ yvf)))
          (t nil))
    (or (and (string= fov "x")
             (fix= xvf 2)
             (fix= yvf 2))
        (and (string= fov "l")
             (fix= xvf 1)
             (fix= yvf 2))
        (and (string= fov "r")
             (fix= xvf 3)
             (fix= yvf 2))
        (and (string= fov "b")
             (fix= xvf 2)
             (fix= yvf 1))
        (and (string= fov "f")
             (fix= xvf 2)
             (fix= yvf 3)))))
```

relationship-vf-hand-fov

```
;; relate detailed visual field to coarse visual field via hand movement
(defun relationship-fov-hand-vf (fov hand vf)
  (let ((xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2))))
    (declare (fixnum xvf yvf))
    (cond ((string= hand "l") (setf xvf (fix1+ xvf)))
          ((string= hand "r") (setf xvf (fix1- xvf)))
          ((string= hand "b") (setf yvf (fix1+ yvf)))
          ((string= hand "f") (setf yvf (fix1- yvf)))
          (t nil))
    (or (and (string= fov "x")
             (fix= xvf 2)
             (fix= yvf 2))
        (and (string= fov "l")
             (fix= xvf 1)
             (fix= yvf 2))
        (and (string= fov "r")
             (fix= xvf 3)
             (fix= yvf 2))
        (and (string= fov "b")
             (fix= xvf 2)
             (fix= yvf 1))
        (and (string= fov "f")
             (fix= xvf 2)
             (fix= yvf 3)))))
```

relationship-fov-hand-vf

relationship-vp-eye-vf

```
;; express the relationship between gaze position, shift and the
;; coarse visual appearance of the body
;; in vp00 the body appears at vf31
;; in vp11 the body appears at vf20
;; so, the first digits must add to 3
;; the second digits must add to 1
;; modulo the eye movement
;; if "l", subtract 1 from xvp
;; if "r", add 1 to xvp
;; if "b", subtract 1 from yvp
;; if "f", add 1 to yvp
;; xvp and yvp range from 0-2 inclusive
(defun relationship-vp-eye-vf (vp eye vf)
  (let ((xvp (read-from-string (subseq vp 0 1)))
        (yvp (read-from-string (subseq vp 1 2)))
        (xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2))))
    (declare (fixnum xvp yvp xvf yvf))
    (cond ((and (string= eye "l") (fix/= 0 xvp)) (setf xvp (fix1- xvp)))
          ((and (string= eye "r") (fix/= 2 xvp)) (setf xvp (fix1+ xvp)))
          ((and (string= eye "b") (fix/= 0 yvp)) (setf yvp (fix1- yvp)))
          ((and (string= eye "f") (fix/= 2 yvp)) (setf yvp (fix1+ yvp)))
          (t nil))
    (and (fix= (fix+ xvp xvf) 3)
         (fix= (fix+ yvp yvf) 1))))
```

relationship-vp-eye-fov

```
;; express the relationship between gaze position, shift and the
;; detailed visual appearance of the body
;; same as above for calculating new vp position
;; then the foveal regions can be mapped easily to the body position
;; note that the body only appears in the back foveal region
(defun relationship-vp-eye-fov (vp eye fov)
  (let ((xvp (read-from-string (subseq vp 0 1)))
        (yvp (read-from-string (subseq vp 1 2))))
    (declare (fixnum xvp yvp))
    (cond ((and (string= eye "l") (fix/= 0 xvp)) (setf xvp (fix1- xvp)))
          ((and (string= eye "r") (fix/= 2 xvp)) (setf xvp (fix1+ xvp)))
          ((and (string= eye "b") (fix/= 0 yvp)) (setf yvp (fix1- yvp)))
          ((and (string= eye "f") (fix/= 2 yvp)) (setf yvp (fix1+ yvp)))
          (t nil))
    (and (string= fov "b")
             (fix= xvp 1)
             (fix= yvp 0))))
```

```
;;; functions used by analyze to express microworld relationships continued

;; relates detailed body visual details to visual position
;; body in b foveal position -> vp10
```

relationship-fov-eye-vp

```
(defun relationship-fov-eye-vp (fov eye vp)
  (and (string= fov "b") ;vp10
       (or (and (string= eye "b")
                (string= vp "10"))
           (and (string= eye "f")
                (string= vp "11"))
           (and (string= eye "l")
                (string= vp "00"))
           (and (string= eye "r")
                (string= vp "20")))))

;; visual network
```

relationship-vp-eye-vp

```
(defun relationship-vp-eye-vp (vp1 eye vp2)
  (let ((x1 (read-from-string (subseq vp1 0 1)))
        (y1 (read-from-string (subseq vp1 1 2)))
        (x2 (read-from-string (subseq vp2 0 1)))
        (y2 (read-from-string (subseq vp2 1 2))))
    (declare (fixnum x1 y1 x2 y2))
    (or (and (string= vp1 vp2)
             (or (and (string= eye "b")
                      (fix= y1 0))
                 (and (string= eye "f")
                      (fix= y1 2))
                 (and (string= eye "l")
                      (fix= x1 0))
                 (and (string= eye "r")
                      (fix= x1 2))))
        (and (string= eye "l")
             (fix= x1 (fix1+ x2))
             (fix= y1 y2))
        (and (string= eye "r")
             (fix= (fix1+ x1) x2)
             (fix= y1 y2))
        (and (string= eye "b")
             (fix= x1 x2)
             (fix= y1 (fix1+ y2)))
        (and (string= eye "f")
             (fix= x1 x2)
             (fix= (fix1+ y1) y2)))))
```

;;; functions used by analyze to express microworld relationships continued

;; hand network

relationship-hp-hand-hp

```
(defun relationship-hp-hand-hp (hp1 hand hp2)
  (let ((x1 (read-from-string (subseq hp1 0 1)))
        (y1 (read-from-string (subseq hp1 1 2)))
        (x2 (read-from-string (subseq hp2 0 1)))
        (y2 (read-from-string (subseq hp2 1 2))))
    (declare (fixnum x1 y1 x2 y2))
    (or (and (string= hp1 hp2)
             (or (and (string= hand "b")
                      (fix= y1 0))
                 (and (string= hand "f")
                      (fix= y1 2))
                 (and (string= hand "l")
                      (fix= x1 0))
                 (and (string= hand "r")
                      (fix= x1 2))))
        (and (string= hand "l")
             (fix= x1 (fix1+ x2))
             (fix= y1 y2))
        (and (string= hand "r")
             (fix= (fix1+ x1) x2)
             (fix= y1 y2))
        (and (string= hand "b")
             (fix= x1 x2)
             (fix= y1 (fix1+ y2)))
        (and (string= hand "f")
             (fix= x1 x2)
             (fix= (fix1+ y1) y2)))))
```

```
;;; functions used by analyze to express microworld relationships continued

;; relates the current body position to a visual position
```

relationship-vf-eye-vp

```
(defun relationship-vf-eye-vp (vf eye vp)
  (let ((xvf (read-from-string (subseq vf 0 1)))
        (yvf (read-from-string (subseq vf 1 2)))
        (xvp (read-from-string (subseq vp 0 1)))
        (yvp (read-from-string (subseq vp 1 2))))
    (declare (fixnum xvp yvp xvf yvf))
    (cond ((string= eye "l") (setf xvf (fix1+ xvf)))
          ((string= eye "r") (setf xvf (fix1- xvf)))
          ((string= eye "b") (setf yvf (fix1+ yvf)))
          ((string= eye "f") (setf yvf (fix1- yvf)))
          (t nil))
    (or (and (fix= (fix+ xvp xvf) 3)
             (fix= (fix+ yvp yvf) 1))
        (and (string= eye "l")
             (fix= xvp 0)
             (fix= (fix+ yvp yvf) 1)
             (string= (subseq vf 0 1) "3"))
        (and (string= eye "r")
             (fix= xvp 2)
             (fix= (fix+ yvp yvf) 1)
             (string= (subseq vf 0 1) "1"))
        (and (string= eye "b")
             (fix= yvp 0)
             (fix= (fix+ xvp xvf) 3)
             (string= (subseq vf 1 2) "1")))))

;; positions where the hand will end up at hp10 in front of the body
```

relationship-hp-hand-body

```
(defun relationship-hp-hand-body (hp hand)
  (or (and (string= hand "b")
           (or (string= hp "10")
               (string= hp "11")))
      (and (string= hand "r")
           (string= hp "00"))
      (and (string= hand "l")
           (string= hp "20"))))

;; positions where the hand moves from hp10 in front of the body
```

relationship-body-hand-hp

```
(defun relationship-body-hand-hp (hand hp)
  (or (and (string= hand "b")
           (string= hp "10"))
      (and (string= hand "f")
           (string= hp "11"))
      (and (string= hand "r")
           (string= hp "20"))
      (and (string= hand "l")
           (string= hp "00"))))
```

```
;;; function used for generating the summary calculated data for each
;;; of the table series
```

calc

```
(defun calc (nums-with-ands)
    (let* ((num1 (delete '& nums-with-ands))
           (num2 (delete '\\ num1))
           (num3 (delete '\\& num2))
           (num4 (delete '\hline num3))
           (nums (delete '\hline& num4))
           (num (length nums))
           (sorted (sort nums #'<)))
      (format t "& ~D & ~D & ~D & ~F ~%"
              (first sorted)
              (nth (fix1- num) sorted)
              (nth (fix1- (floor (fix/ num 2))) sorted)
              (let ((total 0))
                (declare (fixnum total))
                (dolist
                 (x nums)
                 (setf total (fix+ total x)))
                (float (/ total num))))))
```

Appendix B

Output

B.1 Partial output from sample run

```
;;; this is an edited version of one of the test runs of the
;;; reimplementation code, in particular, this output is only through
;;; clock tick 2300 and the numbers of reliable schemas which are
;;; outputted every 50 clock ticks have been removed for readability

 107 spinoff-result  *hcl    /grasp/ -> /grasp/hcl
 263 composite-action-item-positive-created 139 hcl
 327 spinoff-result  .hcl    /ungrasp/ -> /ungrasp/-hcl
 376 spinoff-result  .hp02   /handb/ -> /handb/-hp02
 389 spinoff-result  *vf22   /eyel/ -> /eyel/vf22
 390 spinoff-result  *fovx01 /eyel/ -> /eyel/fovx01
 391 spinoff-result  *fovx02 /eyel/ -> /eyel/fovx02
 392 spinoff-result  *fovx10 /eyel/ -> /eyel/fovx10
 393 spinoff-result  *fovx13 /eyel/ -> /eyel/fovx13
 394 spinoff-result  *fovx23 /eyel/ -> /eyel/fovx23
 395 spinoff-result  *fovx31 /eyel/ -> /eyel/fovx31
 396 spinoff-result  *fovx32 /eyel/ -> /eyel/fovx32
 425 spinoff-result  *hp01   /handb/ -> /handb/hp01
 430 spinoff-result  .vp22   /eyeb/ -> /eyeb/-vp22
 436 spinoff-result  *vf20   /eyel/ -> /eyel/vf20
 446 spinoff-result  *vp11   /eyel/ -> /eyel/vp11
 447 spinoff-result  .vf10   /eyel/ -> /eyel/-vf10
 472 spinoff-result  .vp21   /eyel/ -> /eyel/-vp21
 473 spinoff-result  .tactl  /handr/ -> /handr/-tactl
 474 spinoff-result  .text2  /handr/ -> /handr/-text2
 475 spinoff-result  .text3  /handr/ -> /handr/-text3
 504 spinoff-result  .vp02   /eyer/ -> /eyer/-vp02
 505 spinoff-result  .vf42   /eyer/ -> /eyer/-vf42
 515 spinoff-result  *vp02   /eyel/ -> /eyel/vp02
 516 spinoff-result  .vp02   /eyeb/ -> /eyeb/-vp02
 528 spinoff-result  *hp02   /handf/ -> /handf/hp02
 533 spinoff-result  *hp00   /handb/ -> /handb/hp00
 537 spinoff-result  *vf31   /eyeb/ -> /eyeb/vf31
 545 spinoff-result  .vf02   /eyel/ -> /eyel/-vf02
 560 spinoff-result  *vf42   /eyel/ -> /eyel/vf42
 604 spinoff-result  *vp12   /eyer/ -> /eyer/vp12
 607 spinoff-result  .vf30   /eyer/ -> /eyer/-vf30
 611 spinoff-result  .hp10   /handl/ -> /handl/-hp10
 612 spinoff-result  .tactb  /handl/ -> /handl/-tactb
 613 spinoff-result  .bodyf  /handl/ -> /handl/-bodyf
 614 spinoff-result  .taste0 /handl/ -> /handl/-taste0
 615 spinoff-result  .taste1 /handl/ -> /handl/-taste1
 616 spinoff-result  .taste2 /handl/ -> /handl/-taste2
 617 spinoff-result  *vp21   /eyeb/ -> /eyeb/vp21
 618 spinoff-result  *vf10   /eyeb/ -> /eyeb/vf10
 645 spinoff-result  .hp00   /handr/ -> /handr/-hp00
 655 spinoff-result  *vf03   /eyer/ -> /eyer/vf03
 661 spinoff-result  .vf03   /eyef/ -> /eyef/-vf03
 692 spinoff-result  .vp22   /eyel/ -> /eyel/-vp22
 693 spinoff-result  *vf12   /eyel/ -> /eyel/vf12
```

```
694 spinoff-result  *fovl01 /eyel/ -> /eyel/fovl01
695 spinoff-result  *fovl02 /eyel/ -> /eyel/fovl02
696 spinoff-result  *fovl10 /eyel/ -> /eyel/fovl10
697 spinoff-result  *hp00   /handl/ -> /handl/hp00
698 spinoff-result  *vp12   /eyel/ -> /eyel/vp12
699 spinoff-result  *fovl13 /eyel/ -> /eyel/fovl13
700 spinoff-result  *fovl20 /eyel/ -> /eyel/fovl20
701 spinoff-result  *fovl23 /eyel/ -> /eyel/fovl23
702 spinoff-result  *fovl31 /eyel/ -> /eyel/fovl31
703 spinoff-result  .fovl20 /eyer/ -> /eyer/-fovl20
704 spinoff-result  *fovl32 /eyel/ -> /eyel/fovl32
705 spinoff-result  *hp01   /handf/ -> /handf/hp01
718 spinoff-result  .hp10   /handf/ -> /handf/-hp10
719 spinoff-result  .hp11   /handf/ -> /handf/-hp11
720 spinoff-result  .tactb  /handf/ -> /handf/-tactb
721 spinoff-result  .bodyf  /handf/ -> /handf/-bodyf
722 spinoff-result  .taste0 /handf/ -> /handf/-taste0
723 spinoff-result  .taste1 /handf/ -> /handf/-taste1
724 spinoff-result  .taste2 /handf/ -> /handf/-taste2
729 spinoff-result  *tactf  /handf/ -> /handf/tactf
741 spinoff-result  .vp01   /eyeb/ -> /eyeb/-vp01
742 spinoff-result  .vf30   /eyeb/ -> /eyeb/-vf30
746 spinoff-result  *vf34   /eyer/ -> /eyer/vf34
754 spinoff-result  .vf24   /eyef/ -> /eyef/-vf24
760 spinoff-result  *hp10   /handr/ -> /handr/hp10
761 spinoff-result  *tactb  /handr/ -> /handr/tactb
762 spinoff-result  *bodyf  /handr/ -> /handr/bodyf
763 spinoff-result  *taste0 /handr/ -> /handr/taste0
764 spinoff-result  *taste1 /handr/ -> /handr/taste1
765 spinoff-result  *taste2 /handr/ -> /handr/taste2
771 spinoff-result  .vp00   /eyef/ -> /eyef/-vp00
774 spinoff-result  .vf33   /eyef/ -> /eyef/-vf33
776 spinoff-result  .fovl20 /eyeb/ -> /eyeb/-fovl20
790 spinoff-result  *vp00   /eyeb/ -> /eyeb/vp00
813 spinoff-result  *vf30   /eyef/ -> /eyef/vf30
814 spinoff-result  *vf23   /eyef/ -> /eyef/vf23
815 spinoff-result  *vf43   /eyef/ -> /eyef/vf43
816 spinoff-result  *fovf01 /eyef/ -> /eyef/fovf01
817 spinoff-result  *fovf23 /eyef/ -> /eyef/fovf23
823 spinoff-result  *hp12   /handf/ -> /handf/hp12
824 spinoff-result  *tactl  /handf/ -> /handf/tactl
825 spinoff-result  *text2  /handf/ -> /handf/text2
826 spinoff-result  *text3  /handf/ -> /handf/text3
827 spinoff-result  .vp01   /eyer/ -> /eyer/-vp01
828 spinoff-result  *fovf00 /eyer/ -> /eyer/fovf00
829 spinoff-result  .tactl  /handb/ -> /handb/-tactl
830 spinoff-result  .text2  /handb/ -> /handb/-text2
831 spinoff-result  .text3  /handb/ -> /handb/-text3
832 spinoff-result  .hp01   /handr/ -> /handr/-hp01
```

```
833 spinoff-result  .hp11   /handl/ -> /handl/-hp11
834 spinoff-result  *fovf11 /eyer/ -> /eyer/fovf11
835 spinoff-result  *fovf12 /eyer/ -> /eyer/fovf12
836 spinoff-result  *fovf22 /eyer/ -> /eyer/fovf22
837 spinoff-result  *fovf33 /eyer/ -> /eyer/fovf33
859 spinoff-result  *hp11   /handb/ -> /handb/hp11
864 spinoff-result  *hgr    /grasp/ -> /grasp/hgr
873 spinoff-result  .vp11   /eyer/ -> /eyer/-vp11
874 spinoff-result  *vf14   /eyeb/ -> /eyeb/vf14
881 spinoff-result  .hp12   /handb/ -> /handb/-hp12
882 spinoff-result  .tactr  /handb/ -> /handb/-tactr
885 spinoff-result  *vf03   /eyel/ -> /eyel/vf03
888 spinoff-result  .vf44   /eyef/ -> /eyef/-vf44
890 spinoff-result  *vf13   /eyeb/ -> /eyeb/vf13
892 spinoff-result  .vf13   /eyef/ -> /eyef/-vf13
893 spinoff-result  .hp20   /handf/ -> /handf/-hp20
904 spinoff-result  *vf12   /eyef/ -> /eyef/vf12
905 spinoff-result  .vp21   /eyef/ -> /eyef/-vp21
906 spinoff-result  *fovl01 /eyef/ -> /eyef/fovl01
907 spinoff-result  *fovl02 /eyef/ -> /eyef/fovl02
908 spinoff-result  *fovl10 /eyef/ -> /eyef/fovl10
909 spinoff-result  *fovl13 /eyef/ -> /eyef/fovl13
910 spinoff-result  *fovl23 /eyef/ -> /eyef/fovl23
911 spinoff-result  *fovl31 /eyef/ -> /eyef/fovl31
912 spinoff-result  *fovl32 /eyef/ -> /eyef/fovl32
924 spinoff-result  *tactl  /handl/ -> /handl/tactl
925 spinoff-result  *text2  /handl/ -> /handl/text2
926 spinoff-result  *text3  /handl/ -> /handl/text3
927 spinoff-result  *vf24   /eyeb/ -> /eyeb/vf24
930 spinoff-result  .vf23   /eyeb/ -> /eyeb/-vf23
931 spinoff-result  .fovf01 /eyeb/ -> /eyeb/-fovf01
932 spinoff-result  .fovf23 /eyeb/ -> /eyeb/-fovf23
938 spinoff-result  .vp20   /eyef/ -> /eyef/-vp20
964 spinoff-result  .tactf  /handb/ -> /handb/-tactf
970 spinoff-result  *hp21   /handf/ -> /handf/hp21
974 spinoff-result  *tactr  /handf/ -> /handf/tactr
978 spinoff-result  *fovf00 /eyef/ -> /eyef/fovf00
979 spinoff-result  *fovf11 /eyef/ -> /eyef/fovf11
980 spinoff-result  *fovf12 /eyef/ -> /eyef/fovf12
981 spinoff-result  *fovf22 /eyef/ -> /eyef/fovf22
982 spinoff-result  .vp11   /eyef/ -> /eyef/-vp11
983 spinoff-result  *fovf33 /eyef/ -> /eyef/fovf33
984 spinoff-context  *vp21   /eyel/vf20 -> vp21/eyel/vf20
985 spinoff-context  *vp21   /eyel/vp11 -> vp21/eyel/vp11
988 spinoff-context  *hp01   /handb/hp00 -> hp01/handb/hp00
996 spinoff-result  *vp10   /eyer/ -> /eyer/vp10
997 spinoff-result  *fovb20 /eyer/ -> /eyer/fovb20
```

```
1004 spinoff-result  *vf20   /eyer/ -> /eyer/vf20
1018 spinoff-result  *tactf  /handr/ -> /handr/tactf
1051 spinoff-result  *fovx00 /eyel/ -> /eyel/fovx00
1052 spinoff-result  *fovx11 /eyel/ -> /eyel/fovx11
1053 spinoff-result  *fovx12 /eyel/ -> /eyel/fovx12
1054 spinoff-result  *fovx22 /eyel/ -> /eyel/fovx22
1055 spinoff-result  *fovx33 /eyel/ -> /eyel/fovx33
1068 spinoff-result  .vf11   /eyel/ -> /eyel/-vf11
1069 spinoff-result  *vf33   /eyel/ -> /eyel/vf33
1070 spinoff-result  *vp01   /eyef/ -> /eyef/vp01
1073 spinoff-result  .tactf  /handl/ -> /handl/-tactf
1075 spinoff-result  .vp12   /eyeb/ -> /eyeb/-vp12
1078 spinoff-result  .vf32   /eyer/ -> /eyer/-vf32
1079 spinoff-result  *hp01   /handl/ -> /handl/hp01
1080 spinoff-result  .vf43   /eyer/ -> /eyer/-vf43
1081 spinoff-result  .fovr00 /eyer/ -> /eyer/-fovr00
1082 spinoff-result  .vf43   /eyeb/ -> /eyeb/-vf43
1083 spinoff-result  .fovr01 /eyer/ -> /eyer/-fovr01
1084 spinoff-result  .fovr11 /eyer/ -> /eyer/-fovr11
1085 spinoff-result  .fovr12 /eyer/ -> /eyer/-fovr12
1119 spinoff-context  *fovl00 /eyel/vf22 -> fovl00/eyel/vf22
1120 spinoff-result  .vf44   /eyer/ -> /eyer/-vf44
1121 spinoff-result  .hp00   /handf/ -> /handf/-hp00
1122 spinoff-context  *fovl00 /eyel/fovx01 -> fovl00/eyel/fovx01
1123 spinoff-context  *fovl00 /eyel/fovx02 -> fovl00/eyel/fovx02
1124 spinoff-context  *fovl00 /eyel/fovx10 -> fovl00/eyel/fovx10
1125 spinoff-context  *fovl00 /eyel/fovx13 -> fovl00/eyel/fovx13
1126 spinoff-context  *fovl00 /eyel/fovx23 -> fovl00/eyel/fovx23
1127 spinoff-context  *fovl00 /eyel/fovx31 -> fovl00/eyel/fovx31
1128 spinoff-context  *fovl00 /eyel/fovx32 -> fovl00/eyel/fovx32
1140 spinoff-result  .vf10   /eyef/ -> /eyef/-vf10
1157 spinoff-result  *vp22   /eyer/ -> /eyer/vp22
1158 spinoff-result  .fovr22 /eyer/ -> /eyer/-fovr22
1159 spinoff-result  .fovr23 /eyer/ -> /eyer/-fovr23
1160 spinoff-result  *vf11   /eyeb/ -> /eyeb/vf11
1161 spinoff-result  .fovr33 /eyer/ -> /eyer/-fovr33
1167 spinoff-result  *hp20   /handb/ -> /handb/hp20
1174 spinoff-result  .vf02   /eyeb/ -> /eyeb/-vf02
1184 spinoff-result  *vf23   /eyer/ -> /eyer/vf23
1185 spinoff-result  *fovf01 /eyer/ -> /eyer/fovf01
1186 spinoff-result  *fovf23 /eyer/ -> /eyer/fovf23
1193 spinoff-result  .hp02   /handr/ -> /handr/-hp02
1206 spinoff-context  *hp02   /handb/hp01 -> hp02/handb/hp01
1208 spinoff-context  *hp00   /handr/hp10 -> hp00/handr/hp10
1209 spinoff-context  *hp00   /handr/tactb -> hp00/handr/tactb
1210 spinoff-result  *vp02   /eyef/ -> /eyef/vp02
1211 spinoff-context  *hp00   /handr/bodyf -> hp00/handr/bodyf
1212 spinoff-context  *hp00   /handr/taste0 -> hp00/handr/taste0
1213 spinoff-context  *hp00   /handr/taste1 -> hp00/handr/taste1
```

```
1214 spinoff-context  *hp00   /handr/taste2 -> hp00/handr/taste2
1223 spinoff-result  *fovx03 /eyel/ -> /eyel/fovx03
1224 spinoff-result  *fovx30 /eyel/ -> /eyel/fovx30
1240 spinoff-result  .vp00   /eyer/ -> /eyer/-vp00
1241 spinoff-result  *vf33   /eyer/ -> /eyer/vf33
1243 spinoff-result  *vp21   /eyer/ -> /eyer/vp21
1246 spinoff-result  *vp20   /eyeb/ -> /eyeb/vp20
1250 spinoff-result  .hp21   /handl/ -> /handl/-hp21
1260 spinoff-result  .hp12   /handl/ -> /handl/-hp12
1271 spinoff-result  .vf34   /eyef/ -> /eyef/-vf34
1272 spinoff-result  *fovl20 /eyef/ -> /eyef/fovl20
1288 spinoff-result  .vf34   /eyel/ -> /eyel/-vf34
1289 spinoff-result  *vf42   /eyef/ -> /eyef/vf42
1290 spinoff-result  *vf33   /eyeb/ -> /eyeb/vf33
1305 spinoff-result  .fovf00 /eyeb/ -> /eyeb/-fovf00
1306 spinoff-result  .fovf11 /eyeb/ -> /eyeb/-fovf11
1307 spinoff-result  .fovf12 /eyeb/ -> /eyeb/-fovf12
1308 spinoff-result  .fovf22 /eyeb/ -> /eyeb/-fovf22
1310 syn item created 0 [/handr/-tactl]
1310 syn item created 1 [/handr/-text2]
1310 syn item created 2 [/handr/-text3]
1313 spinoff-context  .hp21   /handl/-hp11 -> -hp21/handl/-hp11
1314 spinoff-result  .vf14   /eyef/ -> /eyef/-vf14
1317 spinoff-result  *hp11   /handr/ -> /handr/hp11
1345 spinoff-result  .hp20   /handl/ -> /handl/-hp20
1346 spinoff-context  *vf24   /eyef/vf23 -> vf24/eyef/vf23
1347 spinoff-context  *vf24   /eyef/fovf01 -> vf24/eyef/fovf01
1348 spinoff-context  *vf24   /eyef/fovf23 -> vf24/eyef/fovf23
1350 spinoff-result  *vp22   /eyef/ -> /eyef/vp22
1361 spinoff-result  .vp20   /eyel/ -> /eyel/-vp20
1369 spinoff-result  *vf13   /eyer/ -> /eyer/vf13
1370 spinoff-result  *fovf03 /handr/ -> /handr/fovf03
1371 spinoff-result  *fovf30 /handr/ -> /handr/fovf30
1380 spinoff-context  *vp01   /eyeb/vf31 -> vp01/eyeb/vf31
1402 spinoff-result  .vf31   /eyer/ -> /eyer/-vf31
1403 spinoff-result  *vf24   /eyer/ -> /eyer/vf24
1404 spinoff-result  .fovf02 /eyer/ -> /eyer/-fovf02
1405 spinoff-result  .fovf10 /eyer/ -> /eyer/-fovf10
1406 spinoff-result  .fovf13 /eyer/ -> /eyer/-fovf13
1407 spinoff-result  .fovf31 /eyer/ -> /eyer/-fovf31
1408 spinoff-result  .fovf32 /eyer/ -> /eyer/-fovf32
1413 spinoff-result  *vp10   /eyel/ -> /eyel/vp10
1414 spinoff-result  *fovb20 /eyel/ -> /eyel/fovb20
1425 spinoff-result  .vf11   /eyef/ -> /eyef/-vf11
1431 spinoff-result  *vf21   /eyel/ -> /eyel/vf21
1432 spinoff-result  *fovb00 /eyel/ -> /eyel/fovb00
1433 spinoff-result  *fovb01 /eyel/ -> /eyel/fovb01
1434 spinoff-result  *fovb02 /eyel/ -> /eyel/fovb02
1435 spinoff-result  *fovb03 /eyel/ -> /eyel/fovb03
```

```
1436 spinoff-result  *fovb10 /eyel/ -> /eyel/fovb10
1437 spinoff-result  *fovb13 /eyel/ -> /eyel/fovb13
1438 spinoff-result  *fovb23 /eyel/ -> /eyel/fovb23
1439 spinoff-result  *fovb30 /eyel/ -> /eyel/fovb30
1440 spinoff-result  *fovb31 /eyel/ -> /eyel/fovb31
1441 spinoff-result  *fovb32 /eyel/ -> /eyel/fovb32
1442 spinoff-result  *fovb33 /eyel/ -> /eyel/fovb33
1443 spinoff-context  *hp00   /handl/hp00 -> hp00/handl/hp00
1456 spinoff-context  .vp20   /eyef/-vp21 -> -vp20/eyef/-vp21
1460 spinoff-result  .hp22   /handb/ -> /handb/-hp22
1463 spinoff-result  .hp21   /handb/ -> /handb/-hp21
1471 spinoff-context  .hp10   /handf/-hp11 -> -hp10/handf/-hp11
1484 spinoff-result  .vf32   /eyeb/ -> /eyeb/-vf32
1485 spinoff-result  .fovr01 /eyeb/ -> /eyeb/-fovr01
1486 spinoff-result  .fovr23 /eyeb/ -> /eyeb/-fovr23
1492 spinoff-result  .vp10   /eyef/ -> /eyef/-vp10
1493 spinoff-result  .fovb20 /eyef/ -> /eyef/-fovb20
1501 spinoff-context  *vp02   /eyel/vp02 -> vp02/eyel/vp02
1506 spinoff-context  *vf33   /eyer/fovf00 -> vf33/eyer/fovf00
1507 spinoff-context  *vf33   /eyer/fovf11 -> vf33/eyer/fovf11
1508 spinoff-context  *vf33   /eyer/fovf12 -> vf33/eyer/fovf12
1526 spinoff-result  .vf21   /eyer/ -> /eyer/-vf21
1527 spinoff-result  .fovb00 /eyer/ -> /eyer/-fovb00
1528 spinoff-result  .fovb01 /eyer/ -> /eyer/-fovb01
1529 spinoff-result  .fovb02 /eyer/ -> /eyer/-fovb02
1530 spinoff-result  .fovb03 /eyer/ -> /eyer/-fovb03
1531 spinoff-result  .fovb10 /eyer/ -> /eyer/-fovb10
1532 spinoff-result  .fovb13 /eyer/ -> /eyer/-fovb13
1533 spinoff-result  .fovb23 /eyer/ -> /eyer/-fovb23
1534 spinoff-result  .fovb30 /eyer/ -> /eyer/-fovb30
1535 spinoff-result  .fovb31 /eyer/ -> /eyer/-fovb31
1536 spinoff-result  .fovb32 /eyer/ -> /eyer/-fovb32
1537 spinoff-result  .fovb33 /eyer/ -> /eyer/-fovb33
1559 spinoff-context  *vp22   /eyeb/vp21 -> vp22/eyeb/vp21
1560 spinoff-context  *vp22   /eyeb/vf10 -> vp22/eyeb/vf10
1561 spinoff-context  *hp11   /handf/hp12 -> hp11/handf/hp12
1564 spinoff-result  .fovf03 /eyer/ -> /eyer/-fovf03
1565 spinoff-result  .fovf30 /eyer/ -> /eyer/-fovf30
1566 spinoff-context  *vf44   /eyer/vf34 -> vf44/eyer/vf34
1573 spinoff-context  *vf12   /eyeb/vf13 -> vf12/eyeb/vf13
1576 spinoff-result  .vf13   /eyel/ -> /eyel/-vf13
1577 spinoff-result  *vf14   /eyel/ -> /eyel/vf14
1583 spinoff-context  *vp01   /eyeb/vp00 -> vp01/eyeb/vp00
1597 spinoff-context  *vf10   /eyel/vf20 -> vf10/eyel/vf20
```

```
1601 spinoff-context  *vf13   /eyer/vf03 -> vf13/eyer/vf03
1608 spinoff-context  *vp22   /eyel/vp12 -> vp22/eyel/vp12
1621 spinoff-result  *hp20   /handr/ -> /handr/hp20
1626 spinoff-context  .hp20   /handl/-hp10 -> -hp20/handl/-hp10
1627 spinoff-result  .vf04   /eyef/ -> /eyef/-vf04
1628 spinoff-context  .hp20   /handl/-tactb -> -hp20/handl/-tactb
1629 spinoff-context  .hp20   /handl/-bodyf -> -hp20/handl/-bodyf
1630 spinoff-context  .hp20   /handl/-taste0 -> -hp20/handl/-taste0
1631 spinoff-context  .hp20   /handl/-taste1 -> -hp20/handl/-taste1
1632 spinoff-context  .hp20   /handl/-taste2 -> -hp20/handl/-taste2
1633 spinoff-result  .vf03   /eyeb/ -> /eyeb/-vf03
1650 spinoff-result  *vf44   /eyeb/ -> /eyeb/vf44
1655 composite-action-item-positive-created 0 hp00
1660 spinoff-context  *hp00   /handf/hp01 -> hp00/handf/hp01
1661 spinoff-context  .vp10   /eyef/-vp11 -> -vp10/eyef/-vp11
1662 spinoff-result  *vf43   /eyel/ -> /eyel/vf43
1663 spinoff-context  *vf10   /eyel/vp11 -> vf10/eyel/vp11
1671 composite-action-item-positive-created 1 hp10
1671 composite-action-item-positive-created 124 tactb
1671 composite-action-item-positive-created 127 bodyf
1671 composite-action-item-positive-created 135 taste0
1671 composite-action-item-positive-created 136 taste1
1671 composite-action-item-positive-created 137 taste2
1672 spinoff-context  *vf30   /eyer/vf20 -> vf30/eyer/vf20
1702 spinoff-result  *fovx02 /handf/ -> /handf/fovx02
1703 spinoff-result  *fovx03 /handf/ -> /handf/fovx03
1704 spinoff-result  *fovx10 /handf/ -> /handf/fovx10
1705 spinoff-result  *fovx13 /handf/ -> /handf/fovx13
1706 spinoff-result  *fovx30 /handf/ -> /handf/fovx30
1707 spinoff-result  *fovx31 /handf/ -> /handf/fovx31
1708 spinoff-result  *fovx32 /handf/ -> /handf/fovx32
1726 spinoff-result  .vf21   /handf/ -> /handf/-vf21
1727 spinoff-result  .fovb00 /handf/ -> /handf/-fovb00
1728 spinoff-result  .fovb01 /handf/ -> /handf/-fovb01
1729 spinoff-result  .fovb02 /handf/ -> /handf/-fovb02
1730 spinoff-result  .fovb03 /handf/ -> /handf/-fovb03
1731 spinoff-result  .fovb10 /handf/ -> /handf/-fovb10
1732 spinoff-result  .fovb13 /handf/ -> /handf/-fovb13
1733 spinoff-result  .fovb23 /handf/ -> /handf/-fovb23
1734 spinoff-result  .fovb30 /handf/ -> /handf/-fovb30
1735 spinoff-result  .fovb31 /handf/ -> /handf/-fovb31
1736 spinoff-result  .fovb32 /handf/ -> /handf/-fovb32
1737 spinoff-result  .fovb33 /handf/ -> /handf/-fovb33
1738 spinoff-result  *vf24   /eyel/ -> /eyel/vf24
1739 spinoff-result  .fovf00 /eyel/ -> /eyel/-fovf00
1740 spinoff-result  .fovf03 /eyel/ -> /eyel/-fovf03
1741 spinoff-result  .fovf11 /eyel/ -> /eyel/-fovf11
1742 spinoff-result  .fovf12 /eyel/ -> /eyel/-fovf12
1743 spinoff-result  .fovf22 /eyel/ -> /eyel/-fovf22
```

```
1744 spinoff-result  .fovf30 /eyel/ -> /eyel/-fovf30
1745 spinoff-result  .vf31   /eyef/ -> /eyef/-vf31
1746 spinoff-result  .fovf33 /eyel/ -> /eyel/-fovf33
1747 spinoff-context  .vf11   /eyef/-vf10 -> -vf11/eyef/-vf10
1749 spinoff-context  *vp00   /eyer/vp10 -> vp00/eyer/vp10
1750 spinoff-result  *fovl00 /handl/ -> /handl/fovl00
1751 spinoff-result  *fovl03 /handl/ -> /handl/fovl03
1752 spinoff-result  *fovl11 /handl/ -> /handl/fovl11
1753 spinoff-result  *fovl12 /handl/ -> /handl/fovl12
1754 spinoff-result  *fovl22 /handl/ -> /handl/fovl22
1755 spinoff-result  *fovl30 /handl/ -> /handl/fovl30
1756 spinoff-result  *fovl33 /handl/ -> /handl/fovl33
1757 spinoff-context  *vp00   /eyer/fovb20 -> vp00/eyer/fovb20
1764 spinoff-result  *vf22   /handf/ -> /handf/vf22
1765 spinoff-result  .fovb11 /handf/ -> /handf/-fovb11
1766 spinoff-result  .fovb12 /handf/ -> /handf/-fovb12
1767 spinoff-result  .fovb22 /handf/ -> /handf/-fovb22
1768 spinoff-result  *fovx00 /handf/ -> /handf/fovx00
1769 spinoff-result  *fovx01 /handf/ -> /handf/fovx01
1770 spinoff-result  *fovx11 /handf/ -> /handf/fovx11
1771 spinoff-result  *fovx12 /handf/ -> /handf/fovx12
1772 spinoff-result  *fovx22 /handf/ -> /handf/fovx22
1773 spinoff-result  *fovx23 /handf/ -> /handf/fovx23
1774 spinoff-result  *fovx33 /handf/ -> /handf/fovx33
1775 spinoff-context  *hp20   /handf/hp21 -> hp20/handf/hp21
1788 spinoff-context  .hp02   /handb/hp00 -> -hp02/handb/hp00
1800 spinoff-context  *vf31   /eyef/vf30 -> vf31/eyef/vf30
1821 spinoff-context  *hp02   /handf/hp02 -> hp02/handf/hp02
1827 spinoff-context  *vp02   /eyer/vp12 -> vp02/eyer/vp12
1844 spinoff-context  *vf23   /eyeb/vf24 -> vf23/eyeb/vf24
1850 spinoff-result  *vf22   /eyef/ -> /eyef/vf22
1851 spinoff-result  *fovx00 /eyef/ -> /eyef/fovx00
1852 spinoff-result  *fovx01 /eyef/ -> /eyef/fovx01
1853 spinoff-result  *fovx02 /eyef/ -> /eyef/fovx02
1854 spinoff-result  *fovx10 /eyef/ -> /eyef/fovx10
1855 spinoff-result  *fovx11 /eyef/ -> /eyef/fovx11
1856 spinoff-result  *fovx12 /eyef/ -> /eyef/fovx12
1857 spinoff-result  *fovx13 /eyef/ -> /eyef/fovx13
1858 spinoff-result  *fovx22 /eyef/ -> /eyef/fovx22
1859 spinoff-result  *fovx23 /eyef/ -> /eyef/fovx23
1860 spinoff-result  *fovx31 /eyef/ -> /eyef/fovx31
1861 spinoff-result  *fovx32 /eyef/ -> /eyef/fovx32
1862 spinoff-result  *fovx33 /eyef/ -> /eyef/fovx33
1863 spinoff-result  .fovr00 /eyeb/ -> /eyeb/-fovr00
1864 spinoff-result  .fovr11 /eyeb/ -> /eyeb/-fovr11
1865 spinoff-result  .fovr12 /eyeb/ -> /eyeb/-fovr12
1866 spinoff-result  .fovr22 /eyeb/ -> /eyeb/-fovr22
1867 spinoff-result  .fovr33 /eyeb/ -> /eyeb/-fovr33
1868 spinoff-result  .fovf20 /eyer/ -> /eyer/-fovf20
1869 spinoff-context  *vf23   /eyer/vf13 -> vf23/eyer/vf13
1870 spinoff-context  *hp12   /handb/hp11 -> hp12/handb/hp11
1874 spinoff-result  *hp22   /handf/ -> /handf/hp22
1892 spinoff-context  *vf44   /eyef/vf43 -> vf44/eyef/vf43
1893 spinoff-context  .vf14   /eyef/-vf13 -> -vf14/eyef/-vf13
```

```
1934 spinoff-context  .vp02   /eyeb/-vp01 -> -vp02/eyeb/-vp01
1935 spinoff-context  .vp02   /eyeb/-vf30 -> -vp02/eyeb/-vf30
1946 spinoff-result  .hp22   /handl/ -> /handl/-hp22
1947 spinoff-context  *tactf  /handf/tactf -> tactf/handf/tactf
1949 spinoff-result  *vf20   /eyef/ -> /eyef/vf20
1971 spinoff-context  *vf43   /eyer/vf33 -> vf43/eyer/vf33
1986 spinoff-result  .tactr  /handl/ -> /handl/-tactr
1991 composite-action-item-positive-created 42 vf44
2010 spinoff-result  *hp21   /handr/ -> /handr/hp21
2013 spinoff-result  .vf42   /handl/ -> /handl/-vf42
2017 spinoff-context  *vp00   /eyef/vp01 -> vp00/eyef/vp01
2020 spinoff-context  .vp01   /eyer/-vp11 -> -vp01/eyer/-vp11
2035 spinoff-context  *vf11   /eyel/vf21 -> vf11/eyel/vf21
2036 spinoff-context  *vf11   /eyel/fovb01 -> vf11/eyel/fovb01
2037 spinoff-context  *vf11   /eyel/fovb02 -> vf11/eyel/fovb02
2038 spinoff-context  *vf11   /eyel/fovb10 -> vf11/eyel/fovb10
2039 spinoff-context  *vf11   /eyel/fovb13 -> vf11/eyel/fovb13
2040 spinoff-context  *vf11   /eyel/fovb23 -> vf11/eyel/fovb23
2041 spinoff-context  *vf11   /eyel/fovb31 -> vf11/eyel/fovb31
2042 spinoff-result  *vp20   /eyer/ -> /eyer/vp20
2043 spinoff-context  *vf11   /eyel/fovb32 -> vf11/eyel/fovb32
2049 spinoff-result  *fovx00 /eyer/ -> /eyer/fovx00
2050 spinoff-result  *fovx11 /eyer/ -> /eyer/fovx11
2051 spinoff-result  *fovx12 /eyer/ -> /eyer/fovx12
2052 spinoff-result  *fovx22 /eyer/ -> /eyer/fovx22
2053 spinoff-result  *fovx33 /eyer/ -> /eyer/fovx33
2061 spinoff-context  *vf11   /eyel/fovb00 -> vf11/eyel/fovb00
2062 spinoff-context  *vf11   /eyel/fovb03 -> vf11/eyel/fovb03
2063 spinoff-context  *vf11   /eyel/fovb30 -> vf11/eyel/fovb30
2064 spinoff-context  *vf11   /eyel/fovb33 -> vf11/eyel/fovb33
2076 spinoff-result  *vf23   /eyel/ -> /eyel/vf23
2077 spinoff-result  *fovf01 /eyel/ -> /eyel/fovf01
2078 spinoff-result  *fovf23 /eyel/ -> /eyel/fovf23
2079 spinoff-context  *vf32   /eyel/vf42 -> vf32/eyel/vf42
2080 spinoff-context  *vf02   /eyel/vf12 -> vf02/eyel/vf12
2081 spinoff-context  *vf02   /eyel/fovl01 -> vf02/eyel/fovl01
2082 spinoff-context  *vf02   /eyel/fovl02 -> vf02/eyel/fovl02
2083 spinoff-context  *vf02   /eyel/fovl10 -> vf02/eyel/fovl10
2084 spinoff-context  *vf02   /eyel/fovl13 -> vf02/eyel/fovl13
2085 spinoff-context  *vf02   /eyel/fovl23 -> vf02/eyel/fovl23
2086 spinoff-context  *vf02   /eyel/fovl31 -> vf02/eyel/fovl31
2087 spinoff-context  *vf02   /eyel/fovl32 -> vf02/eyel/fovl32
```

```
2108 spinoff-result  *hp12   /handr/ -> /handr/hp12
2111 spinoff-context  *vf10   /eyeb/vf11 -> vf10/eyeb/vf11
2116 spinoff-result  *vf44   /eyel/ -> /eyel/vf44
2117 spinoff-context  *vp20   /eyel/vp10 -> vp20/eyel/vp10
2118 spinoff-context  *vf11   /eyel/fovb20 -> vf11/eyel/fovb20
2119 spinoff-result  *vp00   /eyel/ -> /eyel/vp00
2119 composite-action-item-positive-created 13 vp11
2119 composite-action-item-positive-created 20 vf20
2121 spinoff-result  *vf22   /eyer/ -> /eyer/vf22
2122 spinoff-result  *fovx01 /eyer/ -> /eyer/fovx01
2123 spinoff-result  *fovx23 /eyer/ -> /eyer/fovx23
2151 spinoff-context  *vf24   /eyef/fovf00 -> vf24/eyef/fovf00
2152 spinoff-context  *vf24   /eyef/fovf11 -> vf24/eyef/fovf11
2153 spinoff-context  *vf24   /eyef/fovf12 -> vf24/eyef/fovf12
2154 spinoff-context  *vf24   /eyef/fovf22 -> vf24/eyef/fovf22
2155 spinoff-context  *vf24   /eyef/fovf33 -> vf24/eyef/fovf33
2158 spinoff-result  *fovx03 /eyef/ -> /eyef/fovx03
2159 spinoff-result  *fovx30 /eyef/ -> /eyef/fovx30
2160 spinoff-result  *hp02   /handl/ -> /handl/hp02
2168 spinoff-result  .vf21   /eyef/ -> /eyef/-vf21
2169 spinoff-result  .fovb00 /eyef/ -> /eyef/-fovb00
2170 spinoff-result  .fovb01 /eyef/ -> /eyef/-fovb01
2171 spinoff-result  .fovb02 /eyef/ -> /eyef/-fovb02
2172 spinoff-result  .fovb03 /eyef/ -> /eyef/-fovb03
2173 spinoff-result  .fovb10 /eyef/ -> /eyef/-fovb10
2174 spinoff-result  .fovb13 /eyef/ -> /eyef/-fovb13
2175 spinoff-result  .fovb23 /eyef/ -> /eyef/-fovb23
2176 spinoff-result  .fovb30 /eyef/ -> /eyef/-fovb30
2177 spinoff-result  .fovb31 /eyef/ -> /eyef/-fovb31
2178 spinoff-result  .fovb32 /eyef/ -> /eyef/-fovb32
2179 spinoff-result  .fovb33 /eyef/ -> /eyef/-fovb33
2180 spinoff-context  .vf34   /eyef/-vf33 -> -vf34/eyef/-vf33
2192 spinoff-context  *fovf02 /eyef/vf23 -> fovf02/eyef/vf23
2193 spinoff-context  *fovf02 /eyef/fovf01 -> fovf02/eyef/fovf01
2194 spinoff-context  *fovf02 /eyef/fovf23 -> fovf02/eyef/fovf23
```

```
2206 spinoff-result  .fovf20 /eyeb/ -> /eyeb/-fovf20
2207 spinoff-context  *vp11   /eyer/vp21 -> vp11/eyer/vp21
2208 spinoff-context  .vf31   /eyer/-vf21 -> -vf31/eyer/-vf21
2209 spinoff-context  .vf31   /eyer/-fovb00 -> -vf31/eyer/-fovb00
2210 spinoff-context  .vf31   /eyer/-fovb01 -> -vf31/eyer/-fovb01
2211 spinoff-context  .vf31   /eyer/-fovb02 -> -vf31/eyer/-fovb02
2212 spinoff-context  .vf31   /eyer/-fovb03 -> -vf31/eyer/-fovb03
2213 spinoff-context  .vf31   /eyer/-fovb10 -> -vf31/eyer/-fovb10
2214 spinoff-context  .vf31   /eyer/-fovb13 -> -vf31/eyer/-fovb13
2215 spinoff-context  .vf31   /eyer/-fovb23 -> -vf31/eyer/-fovb23
2216 spinoff-context  .vf31   /eyer/-fovb30 -> -vf31/eyer/-fovb30
2217 spinoff-context  .vf31   /eyer/-fovb31 -> -vf31/eyer/-fovb31
2218 spinoff-context  .vf31   /eyer/-fovb32 -> -vf31/eyer/-fovb32
2219 spinoff-context  .vf31   /eyer/-fovb33 -> -vf31/eyer/-fovb33
2223 spinoff-result  *fovf20 /eyel/ -> /eyel/fovf20
2231 composite-action-item-positive-created 34 vf13
2235 spinoff-result  .vp11   /eyeb/ -> /eyeb/-vp11
2239 composite-action-item-positive-created 35 vf23
2239 composite-action-item-positive-created 40 vf24
2239 composite-action-item-positive-created 44 fovf01
2239 composite-action-item-positive-created 54 fovf23
2271 spinoff-context  *hp11   /handl/hp01 -> hp11/handl/hp01
2272 spinoff-context  *vf14   /eyeb/vp20 -> vf14/eyeb/vp20
2279 spinoff-context  *hp01   /handr/hp11 -> hp01/handr/hp11
2295 composite-action-item-positive-created 3 hp01
```

B.2 Sample output from analysis program

```
;;; this is an edited version of the output from the analyze.lisp
;;; program, in particular, many schemas were deleted to make the
;;; output reasonable in length, if the output for a category was
;;; modified, this is indicated after the category

;;; this particular output file was generated from the test run which
;;; appears earlier in this appendix
```

```
category  0:  initial schema
             schemas:    10 reliable schemas:    10
category  1:  grasping schema
             schemas:     8 reliable schemas:     6
category  2:  visual shift schema
             schemas:    51 reliable schemas:    49
category  3:  visual shift limit schema
             schemas:    16 reliable schemas:     6
category  4:  foveal shift schema
             schemas:  282 reliable schemas:  164
category  5:  coarse to detailed visual shift schema
             schemas:  198 reliable schemas:  171
category  6:  visual network schema
             schemas:    63 reliable schemas:    49
category  7:  hand network schema
             schemas:    38 reliable schemas:    34
category  8:  negative consequence schema
             schemas:     0 reliable schemas:     0
category  9:  hand to body schema
             schemas:    15 reliable schemas:    14
category 10:  coarse seeing hand motion schema
             schemas:     1 reliable schemas:     1
category 11:  detailed seeing hand motion schema
             schemas:  103 reliable schemas:    79
category 12:  seeing the body via coarse visual items
             schemas:    17 reliable schemas:    16
category 13:  seeing the body via detailed visual items
             schemas:    31 reliable schemas:    29
category 14:  using the body as a visual reference point
             schemas:     8 reliable schemas:     7
category 15:  hand position required for a given hp translation
             schemas:    10 reliable schemas:    10
category 16:  gaze position required for a given vp translation
             schemas:     5 reliable schemas:     5
category 17:  coarse visual item required for a given vf translation
             schemas:    22 reliable schemas:    20
category 18:  coarse visual item required for a given fov translation
             schemas:    71 reliable schemas:    61
category 19:  detailed visual item required for a given vf translation
             schemas:    18 reliable schemas:    12
category 20:  foveal region relationship with coarse visual field
             schemas:    77 reliable schemas:    54
category 21:  breaking up visual region by where objects are seen
             schemas:    53 reliable schemas:    43
category 22:  hand has to be in a given position to touch the body
             schemas:     5 reliable schemas:     5
category 23:  hand cannot push an object out of the way
             schemas:     8 reliable schemas:     6
category 24:  detailed visual item required for a given fov translation
             schemas:    64 reliable schemas:    50
category 25:  hand in front of body and moving against it
             schemas:    20 reliable schemas:    12
category 26:  hand movement relating coarse to detailed visual items
             schemas:    44 reliable schemas:    30
```

```
category  0:  initial schema
time:          0  schema:  /handr/                          rel-idx:  0.995
time:          0  schema:  /handl/                          rel-idx:  0.995
time:          0  schema:  /handf/                          rel-idx:  1.000
time:          0  schema:  /handb/                          rel-idx:  0.995
time:          0  schema:  /eyer/                           rel-idx:  1.000
time:          0  schema:  /eyel/                           rel-idx:  0.995
time:          0  schema:  /eyef/                           rel-idx:  0.995
time:          0  schema:  /eyeb/                           rel-idx:  0.995
time:          0  schema:  /grasp/                          rel-idx:  1.000
time:          0  schema:  /ungrasp/                        rel-idx:  0.995

category  1:  grasping schema
time:      107  schema:  /grasp/hcl                         rel-idx:  0.821
time:      327  schema:  /ungrasp/-hcl                      rel-idx:  0.990
time:      864  schema:  /grasp/hgr                         rel-idx:  0.006
time:     4457  schema:  tactl/grasp/hgr                    rel-idx:  0.000
time:     5533  schema:  /ungrasp/-hgr                      rel-idx:  0.102
time:     5584  schema:  text2/grasp/hgr                    rel-idx:  0.954
time:     6652  schema:  text3/grasp/hgr                    rel-idx:  0.200
time:     9313  schema:  /ungrasp/(-vp22&-hcl)              rel-idx:  0.000

category  2:  visual shift schema
time:     1346  schema:  vf24/eyef/vf23                     rel-idx:  0.971
time:     1566  schema:  vf44/eyer/vf34                     rel-idx:  0.910
time:     1573  schema:  vf12/eyeb/vf13                     rel-idx:  0.449
time:     1597  schema:  vf10/eyel/vf20                     rel-idx:  0.772
time:     1601  schema:  vf13/eyer/vf03                     rel-idx:  0.199
...category was edited...

category  3:  visual shift limit schema
time:     5293  schema:  -vp12&vf44/eyef/vf43               rel-idx:  0.032
time:     6416  schema:  -vp02&vf21/eyef/vf20               rel-idx:  0.000
time:     7513  schema:  -vp21&vf33/eyer/vf23               rel-idx:  0.000
time:     7579  schema:  -vp02&vf21/eyel/vf31               rel-idx:  0.000
time:     7737  schema:  -vp20&vf12/eyeb/vf13               rel-idx:  0.705
...category was edited...

category  4:  foveal shift schema
time:     1122  schema:  fovl00/eyel/fovx01                 rel-idx:  0.812
time:     3161  schema:  fovx02/eyel/fovr22                 rel-idx:  0.067
time:     3162  schema:  fovx02/eyel/fovr33                 rel-idx:  0.067
time:     3167  schema:  fovr02/eyer/fovx00                 rel-idx:  0.674
time:     7250  schema:  fovx00&fovx02/eyel/fovr33          rel-idx:  0.852
...category was edited...
```

```
category  5:  coarse to detailed visual shift schema
time:     1347  schema:  vf34/eyef/fovf01                  rel-idx:  0.971
time:     1506  schema:  vf33/eyer/fovf00                  rel-idx:  0.125
time:     2036  schema:  vf11/eyel/fovb01                  rel-idx:  0.791
time:     6313  schema:  vf12&fovl00/eyel/fovx00           rel-idx:  0.903
time:     6486  schema:  vf22&fovx00/eyel/fovr11           rel-idx:  0.087
...category was edited...

category  6:  visual network schema
time:      985  schema:  vp21/eyel/vp11                    rel-idx:  0.961
time:     1501  schema:  vp02/eyel/vp02                    rel-idx:  0.946
time:     1559  schema:  vp22/eyeb/vp21                    rel-idx:  0.958
time:     4573  schema:  vp00&vf31/eyer/vp10               rel-idx:  0.645
time:     7944  schema:  vp20&vf11/eyer/vp20               rel-idx:  0.825
...category was edited...

category  7:  hand network schema
time:      968  schema:  hp01/handb/hp00                   rel-idx:  0.827
time:     1208  schema:  hp00/handr/hp10                   rel-idx:  0.948
time:     1443  schema:  hp00/handl/hp00                   rel-idx:  0.953
time:     1561  schema:  hp11/handf/hp12                   rel-idx:  0.874
time:     4120  schema:  tactb/handb/hp10                  rel-idx:  0.888
...category was edited...

category  8:  negative consequence schema

category  9:  hand to body schema
time:     1209  schema:  hp00/handr/tactb                  rel-idx:  0.948
time:     1211  schema:  hp00/handr/bodyf                  rel-idx:  0.948
time:     1212  schema:  hp00/handr/taste0                 rel-idx:  0.948
time:     3176  schema:  hp10/handb/tactb                  rel-idx:  0.919
time:     3177  schema:  hp10/handb/bodyf                  rel-idx:  0.919
time:     3340  schema:  hp10/handb/taste0                 rel-idx:  0.932
...category was edited...

category 10:  coarse seeing hand motion schema
time:     6428  schema:  vf12/handr/vf22                   rel-idx:  0.486

category 11:  detailed seeing hand motion schema
time:     2665  schema:  fovb11/handf/fovx02               rel-idx:  0.786
time:     3725  schema:  fovl00/handr/fovx03               rel-idx:  0.516
time:     6073  schema:  fovx02/handl/fovl00               rel-idx:  0.000
time:     7843  schema:  fovf02/handb/fovx00               rel-idx:  0.667
...category was edited...
```

```
category 12:  seeing the body via coarse visual items
time:     984  schema:  vp21/eyel/vf20                rel-idx:  0.961
time:    1380  schema:  vp01/eyeb/vf31                rel-idx:  0.889
time:    2573  schema:  vp01/eyer/vf20                rel-idx:  0.952
time:    2767  schema:  vp00/eyef/vf30                rel-idx:  0.853
...category was edited...

category 13:  seeing the body via detailed visual items
time:    1757  schema:  vp00/eyer/fovb20              rel-idx:  0.908
time:    2561  schema:  vp20/eyel/fovb01              rel-idx:  0.857
time:    4148  schema:  vp10/eyeb/fovb20              rel-idx:  0.017
time:    6761  schema:  vp11/eyeb/fovb20              rel-idx:  0.905
...category was edited...

category 14:  using the body as a visual reference point
time:    2499  schema:  -fovb20/eyef/-vp11            rel-idx:  0.980
time:    4941  schema:  fovb00/eyeb/vp10              rel-idx:  0.180
time:    5268  schema:  -fovb00/eyef/-vp11            rel-idx:  0.914
time:    6670  schema:  -fovb01/eyef/-vp11            rel-idx:  0.015
time:    7430  schema:  fovb20/eyer/vp20              rel-idx:  0.920
time:    7684  schema:  fovb20/eyel/vp00              rel-idx:  0.711
time:    7847  schema:  -fovb02/eyef/-vp11            rel-idx:  0.667
time:    8921  schema:  -fovb03/eyef/-vp11            rel-idx:  0.000

category 15:  hand position required for a given hp translation
time:    1313  schema:  -hp21/handl/-hp11             rel-idx:  0.983
time:    1471  schema:  -hp10/handf/-hp11             rel-idx:  0.976
time:    1626  schema:  -hp20/handl/-hp10             rel-idx:  0.952
time:    2369  schema:  -hp22/handl/-hp12             rel-idx:  0.940
time:    2486  schema:  -tactb/handf/-hp11            rel-idx:  0.973
time:    2785  schema:  -hp22/handb/-hp21             rel-idx:  0.972
time:    3555  schema:  -bodyf/handf/-hp11            rel-idx:  0.976
time:    4369  schema:  -taste0/handf/-hp11           rel-idx:  0.324
time:    4963  schema:  -taste1/handf/-hp11           rel-idx:  0.687
time:    5601  schema:  -taste2/handf/-hp11           rel-idx:  0.895

category 16:  gaze position required for a given vp translation
time:    1456  schema:  -vp20/eyef/-vp21              rel-idx:  0.988
time:    1661  schema:  -vp10/eyef/-vp11              rel-idx:  0.988
time:    1934  schema:  -vp02/eyeb/-vp01              rel-idx:  0.988
time:    2020  schema:  -vp01/eyer/-vp11              rel-idx:  0.987
time:    3482  schema:  -vp12/eyeb/-vp11              rel-idx:  0.977

category 17:  coarse visual item required for a given vf translation
time:    1747  schema:  -vf11/eyef/-vf10              rel-idx:  0.872
time:    2208  schema:  -vf31/eyer/-vf21              rel-idx:  0.857
time:    2332  schema:  -vf22/eyeb/-vf23              rel-idx:  0.895
time:    3414  schema:  -vf01/eyel/-vf11              rel-idx:  0.877
...category was edited...
```

```
category 18:  coarse visual item required for a given fov translation
time:     2209  schema:  -vf31/eyer/-fovb00              rel-idx:  0.857
time:     2333  schema:  -vf22/eyeb/-fovf01              rel-idx:  0.895
time:     2580  schema:  -vf42/eyer/-fovr00              rel-idx:  0.796
time:     5257  schema:  -vf11/eyeb/-fovl20              rel-idx:  0.892
...category was edited...

category 19:  detailed visual item required for a given vf translation
time:     3185  schema:  -fovr02/eyef/-vf31              rel-idx:  0.919
time:     4176  schema:  -fovr03/eyef/-vf31              rel-idx:  0.017
time:     4596  schema:  -fovl00/eyef/-vf11              rel-idx:  0.645
time:     4895  schema:  -fovx01/eyef/-vf21              rel-idx:  0.733
time:     5255  schema:  -fovl01/eyef/-vf11              rel-idx:  0.935
...category was edited...

category 20:  foveal region relationship with coarse visual field
time:     1119  schema:  fovl00/eyel/vf22              rel-idx:  0.812
time:     2510  schema:  fovf01/eyeb/vf24              rel-idx:  0.480
time:     2682  schema:  fovl00/eyeb/vf13              rel-idx:  0.269
time:     3172  schema:  fovr02/eyer/vf22              rel-idx:  0.852
time:     3204  schema:  fovf00/eyef/vf22              rel-idx:  0.597
...category was edited...
```

```
category 21:  breaking up visual region by where objects are seen
time:    1935  schema:  -vp02/eyeb/-vf30              rel-idx:  0.988
time:    2667  schema:  -vp20/eyef/-vf14              rel-idx:  0.931
time:    2668  schema:  -vp21/eyef/-vf13              rel-idx:  0.683
time:    2847  schema:  -vp20/eyef/-vf10              rel-idx:  0.768
time:    2879  schema:  -vp00/eyef/-vf44              rel-idx:  0.972
time:    3328  schema:  -vp00/eyer/-vf21              rel-idx:  0.636
time:    3329  schema:  -vp00/eyer/-fovb00            rel-idx:  0.652
time:    3330  schema:  -vp00/eyer/-fovb01            rel-idx:  0.636
time:    3331  schema:  -vp00/eyer/-fovb02            rel-idx:  0.636
time:    3332  schema:  -vp00/eyer/-fovb03            rel-idx:  0.652
time:    3333  schema:  -vp00/eyer/-fovb10            rel-idx:  0.636
time:    3334  schema:  -vp00/eyer/-fovb13            rel-idx:  0.636
time:    3335  schema:  -vp00/eyer/-fovb23            rel-idx:  0.636
time:    3336  schema:  -vp00/eyer/-fovb30            rel-idx:  0.652
time:    3337  schema:  -vp00/eyer/-fovb31            rel-idx:  0.636
time:    3338  schema:  -vp00/eyer/-fovb32            rel-idx:  0.636
time:    3339  schema:  -vp00/eyer/-fovb33            rel-idx:  0.652
time:    3511  schema:  -vp12/eyeb/-vf20              rel-idx:  0.977
time:    4302  schema:  vp21/eyef/fovf00             rel-idx:  0.000
time:    4423  schema:  vp21/eyef/fovf11             rel-idx:  0.664
time:    4424  schema:  vp21/eyef/fovf12             rel-idx:  0.182
time:    4425  schema:  vp21/eyef/fovf22             rel-idx:  0.955
time:    4426  schema:  vp21/eyef/fovf33             rel-idx:  0.918
time:    4683  schema:  vp01/eyer/vf34               rel-idx:  0.810
time:    5570  schema:  -vp02/eyer/-vf42              rel-idx:  0.103
time:    5600  schema:  vp22/eyel/vf42               rel-idx:  0.954
time:    5640  schema:  -vp01/eyer/-vf43              rel-idx:  0.023
time:    5824  schema:  -vp00/eyer/-vf44              rel-idx:  0.951
time:    5889  schema:  -vp12/eyeb/-vf43              rel-idx:  0.741
time:    6594  schema:  vp10/eyef/vf43               rel-idx:  0.388
time:    6741  schema:  vp21/eyel/vf43               rel-idx:  0.000
time:    7050  schema:  -vp12/eyeb/-vf02              rel-idx:  0.638
time:    7352  schema:  vp20/eyef/fovf11             rel-idx:  0.000
time:    7353  schema:  vp20/eyef/fovf12             rel-idx:  0.000
time:    7354  schema:  vp20/eyef/fovf22             rel-idx:  0.000
time:    7355  schema:  vp20/eyef/fovf33             rel-idx:  0.020
time:    7387  schema:  vp10/eyer/vf34               rel-idx:  0.608
time:    7407  schema:  -vp20/eyef/-vf33              rel-idx:  0.900
time:    7463  schema:  vp12/eyef/vf42               rel-idx:  0.673
time:    7603  schema:  vp20/eyel/vf24               rel-idx:  0.000
time:    7648  schema:  vp21/eyer/vf33               rel-idx:  0.000
time:    7769  schema:  vp11/eyeb/vf44               rel-idx:  0.628
time:    8030  schema:  -vp22/eyer/-vf32              rel-idx:  0.000
time:    8031  schema:  -vp22/eyer/-fovr00            rel-idx:  0.842
time:    8032  schema:  -vp22/eyer/-fovr01            rel-idx:  0.000
time:    8033  schema:  -vp22/eyer/-fovr11            rel-idx:  0.553
time:    8034  schema:  -vp22/eyer/-fovr12            rel-idx:  0.211
time:    8035  schema:  -vp22/eyer/-fovr22            rel-idx:  0.079
time:    8036  schema:  -vp22/eyer/-fovr23            rel-idx:  0.079
time:    8037  schema:  -vp22/eyer/-fovr33            rel-idx:  0.237
time:    8397  schema:  vp20/eyef/vf23               rel-idx:  0.290
time:    8398  schema:  vp20/eyef/fovf01             rel-idx:  0.000
time:    8399  schema:  vp20/eyef/fovf23             rel-idx:  0.097
```

```
category 22:  hand has to be in a given position to touch the body
time:    1628  schema:  -hp20/handl/-tactb          rel-idx:  0.910
time:    1629  schema:  -hp20/handl/-bodyf          rel-idx:  0.946
time:    1630  schema:  -hp20/handl/-taste0         rel-idx:  0.952
time:    1631  schema:  -hp20/handl/-taste1         rel-idx:  0.946
time:    1632  schema:  -hp20/handl/-taste2         rel-idx:  0.946

category 23:  hand cannot push an object out of the way
time:    1947  schema:  tactf/handf/tactf           rel-idx:  0.969
time:    4121  schema:  tactb/handb/tactb           rel-idx:  0.888
time:    4450  schema:  tactl/handl/tactl           rel-idx:  0.000
time:    4451  schema:  tactl/handl/text2           rel-idx:  0.257
time:    4551  schema:  tactl/handl/text3           rel-idx:  0.869
time:    5711  schema:  text2/handl/tactl           rel-idx:  0.929
time:    5918  schema:  tactr/handr/tactr           rel-idx:  0.237
time:    6867  schema:  text3/handl/tactl           rel-idx:  0.000

category 24:  detailed visual item required for a given fov translation
time:    4324  schema:  -fovx00/eyeb/-fovf00        rel-idx:  0.920
time:    4325  schema:  -fovx00/eyeb/-fovf11        rel-idx:  0.893
time:    4563  schema:  -fovx01/eyeb/-fovf23        rel-idx:  0.869
time:    4882  schema:  -fovx00/eyef/-fovb00        rel-idx:  0.010
time:    4883  schema:  -fovx00/eyef/-fovb03        rel-idx:  0.010
...category was edited...

category 25:  hand in front of body and moving against it
time:    4122  schema:  tactb/handb/bodyf           rel-idx:  0.750
time:    6198  schema:  bodyf/handb/taste1          rel-idx:  0.000
time:    6397  schema:  bodyf/handb/tactb           rel-idx:  0.775
time:    7622  schema:  taste0/handb/taste2         rel-idx:  0.761
time:    7792  schema:  taste0/handb/bodyf          rel-idx:  0.791
...category was edited...

category 26:  hand movement relating coarse to detailed visual items
time:    3110  schema:  fovb11/handf/vf22           rel-idx:  0.824
time:    3552  schema:  vf22/handl/fovl00           rel-idx:  0.189
time:    4686  schema:  fovb12/handf/vf22           rel-idx:  0.819
time:    4802  schema:  vf31/handl/fovb11           rel-idx:  0.294
time:    6784  schema:  fovx03/handf/vf23           rel-idx:  0.190
...category was edited...

*** non-categorized non-empty context rel-idx above .7 ***
time:    2611  schema:  -vf30/eyer/-vp11            rel-idx:  0.986
time:    2666  schema:  -vf11/eyef/-vp21            rel-idx:  0.979
time:    2696  schema:  -hp22/handl/-tactf          rel-idx:  0.786
time:    2818  schema:  -vp10/handf/-vf21           rel-idx:  0.817
time:    2819  schema:  -vp10/handf/-fovb00         rel-idx:  0.859
...lots of stuff deleted...
```

Bibliography

[Dre86] Gary Drescher. Genetic ai – translating piaget into lisp. Technical Report A.I. Memo No. 890, Massachusetts Institute of Technology Artificial Intelligence Laboratory, February 1986.

[Dre89] Gary Drescher. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence.* PhD thesis, Massachusetts Institute of Technology Artificial Intelligence Laboratory, September 1989.

[Dre91] Gary Drescher. *Made-Up Minds: A Constructivist Approach to Artificial Intelligence.* MIT Press, Cambridge MA, 1991.

[Mae92] Pattie Maes. Slides from 4.996: Modeling and building autonomous agents. (lecture given on the 19th), March 1992.

[SESA86] Douglas Smith, Maurice Eggen, and Richard St. Andre. *A Transition to Advanced Mathematics.* Brooks/Cole Publishing, Monterey CA, 2nd edition, 1986.