Parallel Retrieval Algorithms For Semantic Nets

by

Hae Jin Baek

SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1986

Signature of Author Signature redacted
Department of Electrical Engineering and Computer Science
March 11, 1986

Certified by Signature redacted
Professor Robert H. Halstead, Jr.
Thesis Supervisor

Accepted by Signature redacted
David Adler
Chairman, Department Committee

Parallel Retrieval Algorithms For Semantic Nets

by

Hae Jin Baek

Submitted to the Department of Electrical Enginerring and Computer Science
on March 11, 1986 in partial fulfillment of the requirements for
the degree of Bachelor of Science in Computer Science

Abstract

Semantic Nets are used in representing complex relations between data. Their use in natural language processing data bases, image recognition networks, and expert systems is constrained by their retrieval times. One way of reducing retrieval times is through the use of parallel machines. One such multiprocessor system, Concert, is currently being developed by the Real Time Systems Group at M.I.T..

This thesis documents the experiments to determine the effects of parallelism on information retrieval times. Various benchmark queries were used on two semantic nets to ascertain this.

The results show that parallelism properly implemented can significantly enhance retrieval times for semantic nets.

Name and Title of Thesis Supervisor:

Robert H. Halstead, Jr.
Assistant Professor of Computer Science and Engineering

Acknowledgements

My deepest gratitude goes to Professor Halstead without whose suggestions, patience, and encouragement, none of this would have been possible. I would like to thank Boris Katz, David Chanen, and Bob Frank, whose work with Semantic Nets offered the starting point for this research. I would also like to thank the members of the CONCERT project for their help with CONCERT.

Warmest thanks also goes to Cary Ching, Rich Roth, and Yona Kaplan, who provided an endless resource of names for my databases.

Table of Contents

Table of Illustrations

Chapter I

Introduction

1.1. Motivation

Semantic Nets are used as the basic framework for large databases in many A.I. fields. They serve to store information in expert systems, natural language parser data bases, and image recognition networks [3, 4, 8]. Because of their complexity and size, processing information can sometimes be a monumental feat. The current retrieval algorithms rely on single-processor machines, resulting in very long processing times. One answer to speeding up processing may come with multiprocessors, which may one day replace single-processor machines because they provide faster performance times. This desire to keep semantic nets up to date with advancing technology has led to this research. The purpose of this research is to make use of the parallel processing techniques available to speed up the processing times for semantic net retrievals.

1.2. The Problem and The Method

There exist many internal representations of a semantic net [2], and there is a

corresponding retrieval algorithm *best* suited for each one. Focusing on one internal representation that is widely used, and identifying suitable opportunities for parallelism in the retrieval algorithms, this research will examine the effects of parallel techniques on the retrieval times.

Currently The Real Time Systems Group of M.I.T.'s Laboratory For Computer Science has a multiprocessor system called CONCERT [1]. CONCERT can be run with any number of processors up to 15. CONCERT runs *MultiLisp* [6], a parallel version of LISP. Multilisp contains a construct called **future** which allows the parallel execution of operations. Calling a **future** of an expression immediately returns a place holder which is replaced by the result of the evaluation when it is completed. This eliminates waiting for an expression to be evaluated before evaluating the next expression, thus allowing many evaluations to occur simultaneously.

Using two example semantic net data bases, different parallel versions of the retrieval algorithms, with **futures** placed in various strategic locations, have been tested on CONCERT. Measurements were taken during the execution of a number of example queries designed to test all possible paths of the algorithms. Each test uses a varying number of processors. The results of the experiment show the effects of parallel processing techniques on these retrieval algorithms.

1.3. Thesis Overview

Chapter 2 introduces the semantic net algorithms and discusses the data structures involved. Chapter 3 examines the algorithms and identifies possible places where futures would be helpful. Chapter 4 presents the results of the experiment. Chapter 5 concludes by discussing the outcome of the experiment.

Chapter II

The Semantic Net Algorithms

The semantic net is used to store a series of symbolic relationships. The retrieval algorithms used for testing were taken from a natural language database, developed by Boris Katz and Patrick Winston at M.I.T.'s Artifical Intelligence Laboratory [7], and is written in Multilisp.

2.1. Data Structures

Each symbolic relationship which consists of a subject, relation, and object:

[*SUBJECT RELATION OBJECT*]

is internally represented as a *link*, which consists of six fields. The fields are *subject*, *relation*, *object*, *subject-of*, *object-of*, and *truth-value* . The *subject*, *relation*, and *object* fields contain the subject, relationship, and object of the link respectively. The *subject-of* and *object-of* fields point to the list of links that this particular link is a subject of and object of respectively. The *truth-value* can be true, false, or unknown according

to how the link is entered. It is true if the relationship is valid, false if invalid, and unknown if there is a question of its validity.

The *subject* and *object* fields can have as its value an atom or another link, thereby allowing nested link structures. The relation can only be an atom. An atom is the basic single object in Lisp.

When a relationship is entered, a link is created with the corresponding subject, relationship, and object in its proper fields. If the subject is another link, then the whole enclosing link is placed in the *subject-of* list of the subject-link. For example the link

$$[\,[SUBJECT1\ REL1\ OBJECT1\,]\ REL2\ OBJECT2\,]$$

has a link as its subject, so the whole *REL2* link will be placed in the *subject-of* list of the subject *REL1* link, as shown in Figure 2.1.

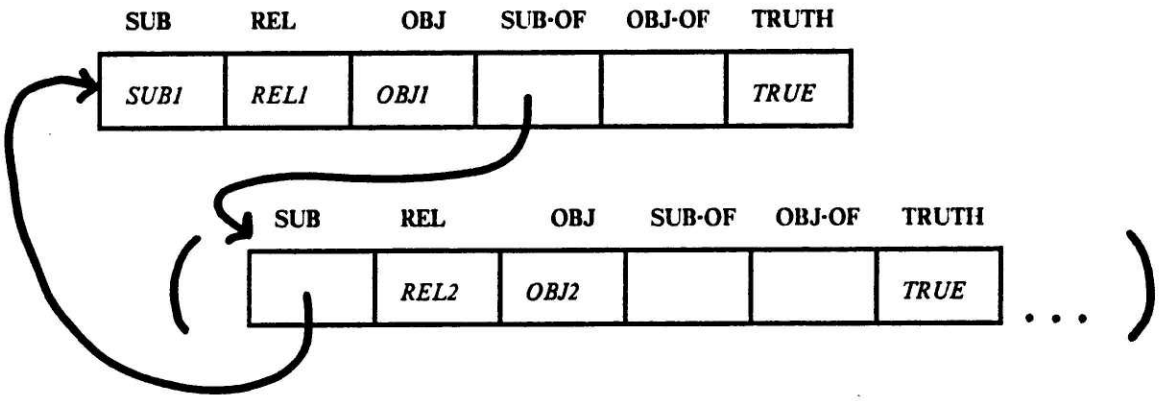


Figure 2.1. Using a link as a subject

If the subject or object is an atom, then associated with the atom is a property list with slots for *subject-of* and *object-of* lists of links. The enclosing link is placed in the correct slot of the subject atom or object atom's property list. Associated with

each relation atom is a property list containing all links that this atom is the relation of. Figure 2.2 illustrates the link representing the relation "*John loves Mary*" and the property list associated with the subject *John*.

[*JOHN LOVES MARY*]

JOHN $\Longrightarrow$ *plist* (*subject-of* : [*JOHN LOVES MARY*] ..)

Figure 2.2. The Atom's Property List

The *subject-of* and *object-of* lists provide a record-keeping system that keeps track of the instances of various atoms, links, and nested structures. This makes processing much easier since the whole net does not have to be searched for a certain value.

2.2. The Retrieval Algorithm

Queries take many forms. Either one or two of the slots can be questioned, *i.e.*, at least one of the slots must be known and it must be an atom or link, but the other slots can be atoms, lists of atoms, or lists of other links. With at least one known atom, the resulting list of links contain specific information about that atom. If the other slots are also atoms or links, then the resulting information is more specific, since more information was known beforehand, the possible range of answers were narrowed down. If the other slots were lists of links, then the resulting list of links would contain information about all the elements in those given links, *i.e.*, it would produce a union of the resulting links produced by successive queries called with each element of the list. For example, all of the following are legal queries:

[??? *LOVES MARY*] [??? ??? *MARY*]

[*JOHN ??? MARY*] [*JOHN ??? ???*]

[*JOHN LOVES ???*] [*??? LOVES ???*]

which would be satisfied by

[*JOHN LOVES MARY*].

The retrieval algorithm first checks if the subject is known, and if it is an atom. If it is, it then finds all links that this atom is a subject of by conveniently extracting the *subject-of* list from the atom's property list. It then filters the *relation* and *object* slots by matching the value given in the query to the ones in the retrieved links, and discarding the links that do not match. Unknowns are treated as wildcards, which match all. The resulting list of link(s) should have the same known values as the query with all unknowns filled in.

If the subject is not known or is not an atom, it then checks the object, then the relation, filtering out the other two slots in the same manner. This simple algorithm is implemented in the **matching-links** function shown in Figure 2.3. The filtering of the other two slots is done by **filter-links**, shown in Figure 2.4, which is called by **matching-links**. **Filter-links** is an intermediary function which takes as arguments *filter*, which is the desired known value, *links*, which is the list of links to search through, and *filter-function*, which is used to extract the correct slot from each link in the list of links. **Filter-links** calls **remove-if-not-equal** or **remove-if-not-member** depending on the type of the argument *filter*; it calls **remove-if-not-member** if *filter* is a list. These two *remove-if-not-xxx* functions, shown in Figure 2.5, handle the two different kinds of matching: equality and membership. **Remove-if-not-equal** tests equality of atoms and **remove-if-not-member** tests membership in lists.

It is important that the relation slot is checked last because it is more likely that a particular relation will exist in more links than a particular subject or object. In other words, the *relation-of* list for a given relation will usually be much larger than a

subject-of or *object-of* list for a given subject or object, which means more matching. Given there is one known atom, if any of the other two slots is a list, then matching is done by checking for the membership of a value (*from the retrieved links*) in this list.

```
(defun matching-links (subject relation object)
    (cond ((and (not-wild subject)
                (or (atom subject) (link-p subject)))
           (filter-links object
                (filter-links relation (subject-of subject) link-relation-fn)
                link-object-fn))
          ((and (not-wild object)
                (or (atom object) (link-p object)))
           (filter-links subject
                (filter-links relation (object-of object) link-relation-fn)
                link-subject-fn))
          ((and (not-wild relation)
                (or (atom relation) (link-p relation)))
           (filter-links object
                (filter-links subject (relation-of relation) link-subject-fn)
                link-object-fn))
          (t (error "At least one of the match arguments must be atomic"))))
```

Figure 2.3. Function **matching-links**

So there are three possible paths down the matching function: (1) the subject is a known atom or link, (2) the object is a known atom or link, or (3) the relation is the only known atom. Each of the three paths filters the other two remaining slots. The first two paths are very similar but the third one may prove to be slower because the

```
(defun filter-links (filter links filter-function)
        (cond ((eq  filter  *wildcard* ) links)
              ((or (atom  filter)  (link-p  filter))
               (remove-if-not-equal  filter links filter-function))
              ((listp filter)
               (remove-if-not-member filter links filter-function))))
```

Figure 2.4. Function **filter-links**

```
(defun remove-if-not-equal (filter links filter-function)
        (cond ((null  links)  nil)
              ((eq  (filter-function (car links))  filter)
               (cons (car links)
                     (remove-if-not-equal filter (cdr links) filter-function)))
              (t  (remove-if-not-equal  filter (cdr links) filter-function))))

(defun remove-if-not-member (filter links filter-function)
        (cond ((null  links)  nil)
              ((member  (filter-function (car links))  filter)
               (cons (car links)
                     (remove-if-not-member filter (cdr links) filter-function)))
              (t  (remove-if-not-member  filter (cdr links) filter-function))))
```

Figure 2.5. The Removal Functions

starting list of links, which result from extracting the *relation-of* list for an atom, will

usually be very large.

The result of **matching-links** is a list of links. It is sometimes necessary to extract the subject or object out of the links depending on what the query asked for. For example, in the case of the parent query: *Who are Fallon's parents?*, what is desired is (*Blake Alexis*) instead of the whole list

([*BLAKE PARENT – OF FALLON*] [*ALEXIS PARENT – OF FALLON*])

The functions for pulling subjects or objects out of a list of links, shown in Figure 2.6, will also provide opportunities for parallelism.

```
(defun pull-subjects (links)
        (if (null  links)
           nil
           (cons (link-subject  (car links))
                 (pull-subjects  (cdr links)))))

(defun pull-objects (links)
        (if (null  links)
           nil
           (cons (link-object  (car links))
                 (pull-objects  (cdr links)))))
```

Figure 2.6. The Pull Functions

2.3. Scenarios

Lets consider a semantic net representing a family tree, with information such as parents and siblings. This semantic net needs only three kinds of relationships because

the other family relationships such as aunts, uncles, grandparents, or cousins can be derived from the relationships *PARENT*, *SIBLING-OF* and *SEX*. The number of links will vary with the size of the family.

Parents are entered as:

[*BLAKE PARENT – OF FALLON*]

[*ALEXIS PARENT – OF FALLON*]

and for asserting fatherhood, motherhood, sisterhood, *etc...*, we assert the sex.

[*BLAKE SEX MALE*]

[*ALEXIS SEX FEMALE*]

[*FALLON SEX FEMALE*]

Siblings as:

[*FALLON SIBLING – OF STEVEN*]

[*ADAM SIBLING – OF STEVEN*]

You don't need a **children** relation because to find the parents of someone, you need only specify the child and the relation *PARENT-OF* .

[??? *PARENT – OF FALLON*]

This eliminates redundant storage of data.

A typical query would be to find someone's grandparents can be represented as

[??? *PARENT – OF* [??? *PARENT – OF FALLON*]]

to find *FALLON* 's grandparents.

The program to do so is :

```
(defun grandparents (child)
    (pull-subjects
      (matching-links *wildcard* PARENT-OF
        (pull-subjects
          (matching-links *wildcard* PARENT-OF child))))
```

2.4. Potential Opportunities For Parallelism

The opportunities for parallelism lie in the matching functions, since all the time is spent filtering through the links. The sequential matching function must match each link one at a time, which amounts to a linear search down the list of links. The ideal parallel version of this function would match all links in parallel, eliminating the need to wait for the end of the linear search.

An obvious location for a future in **remove-if-not-equal** is right before the first recursive call to itself, as shown in Figure 2.7. Since the function returns a cons cell, this future will provide a place holder for the cdr of the cell, and because the results of the recursive **p-remove-if-not-equal** evaluation are not needed right away, the function **p-remove-if-not-equal** will work well.

```
(defun p-remove-if-not-equal (filter links filter-function)
    (cond ((null links) nil)
          ((eq (filter-function (car links)) filter)
           (cons (car links)
                 (future (p-remove-if-not-equal
                           filter (cdr links) filter-function))))
          (t (p-remove-if-not-equal filter (cdr links) filter-function))))
```

Figure 2.7. Parallel Removal Function: Version I

One might also suggest a future before the second recursive call to **p-remove-if-not-equal** but this is not a good idea. This way, the function would return a future whose value was also a chain of futures. It is always a good idea to refrain from returning just a future [5].

Upon closer examination, it is evident that the recursive call to itself happens in

both branches of the conditional, *i.e.*, it happens independently of a successful match. Exploiting this fact, Figure 2.8 provides another way to use futures for this function. In **p2-remove-if-not-equal** the future call works well because the future value will not be touched inside the program. It will either be built in at the end of the resulting list, or returned as the result. Although **p2-remove-if-not-equal** can result in a chain of futures, it is designed to get results back to its caller as soon as possible, even while the matching is still in progress [6].

```
(defun p2-remove-if-not-equal (filter links filter-function)
    (if (null  links)
       nil
       ((let ((rest (future
                        (p2-remove-if-not-equal filter (cdr links) filter-function))))
          (if    (eq (filter-function (car links))  filter)
                 (cons (car links) rest)
                 rest))))
```

Figure 2.8. Parallel Removal Function: Version II

In the functions for testing equality the second parallel version should not make a big difference from the first version because *eq* is a quick operation. This style might make a bigger difference in **remove-if-not-member** because *member* must search a list.

Another place to look for possible parallelism is the place where the matching functions are called. It is a good idea in this case to place a future before the first call to the *remove-if-not-xxx* functions in **filter-links** , as shown in Figure 2.9.

There is also a place for futures in the **pull-subjects** and **pull-objects** functions.

```
(defun p-filter-links (filter links filter-function)
        (cond ((eq  filter *wildcard* ) links)
              ((or (atom  filter)  (link-p  filter))
               (future (remove-if-not-equal  filter links filter-function))
              ((listp  filter)
               (future (remove-if-not-member  filter links filter-function))))
```

Figure 2.9. Parallel Filter Function

Each of these recursive functions also returns a cons cell, the cdr of which is computed by a recursive call to itself. For the reason stated earlier, a future can be placed just before the recursive call, as shown in Figure 2.10.

```
(defun p-pull-subjects (links)
        (if (null  links)
           nil
           (cons (link-subject  (car links))
                 (future  (p-pull-subjects  (cdr links))))))
```

Figure 2.10. Extraction Algorithm: Parallel Version

Chapter III

Experimental Results

The semantic nets used for testing were (1) one representing the family hierarchy as described above and (2) a school database consisting of information such as classes, professors, students, and grades.

The family tree net is an example of a deep network with few kinds of relations but many links. All the links in this network contain atoms for subject and object slots; there are no nested link structures. This network contains 482 objects and 1667 links. Figure 3.1 shows sample entries in this data base. Note that all slots contain single atoms.

These data bases were randomly generated, and saved in a file, so the same data base was used for all versions of the tests.

The school database provides an example of a network with many kinds of relations, offering deeply nested link structures. This net contains 192 objects and 2931 links. This contains many fewer objects than links because it uses the same links over again, as other *subjects* and *objects*. Figure 3.2 shows sample entries in this school data base. Note the nesting of the links as in the *GRADE* and *ROOM* relation links.

[blake sex male] [alexis sex female]

[fallon sex female] [steven sex male]

[danny sex male] [mary sex female]

[jason sex male] [sable sex female]

[monica sex female] [miles sex male]

[jeff sex male] [francesca sex female]

[blake parent-of fallon] [alexis parent-of fallon]

[blake parent-of steven] [alexis parent-of steven]

[fallon sibling-of steven] [steven sibling-of fallon]

[jason parent-of monica] [sable parent-of monica]

[jason parent-of miles] [sable parent-of miles]

[miles sibling-of monica] [monica sibling-of miles]

[jason parent-of jeff] [francesca parent-of jeff]

[fallon parent-of danny] [jeff parent-of danny]

[fallon parent-of mary] [jeff parent-of mary]

[mary sibling-of danny] [danny sibling-of mary]

Figure 3.1. Example of a family data base

3.1. The Benchmarks

To summarize the different versions of the matching algorithms, there are three versions of the *remove-if-not-equal* functions

sequential, p − remove, and p2 − remove,

three versions of the *remove-if-not-member* functions

sequential, p − remove, and p2 − remove,

[[melissa course [algebra term fall85]] grade b]

[[henry course [algebra term fall82]] grade a]

[[susan course [algebra term spr81]] grade f]

[[danny course [algebra term fall86]] grade d]

[[christine course [algebra term spr85]] grade c]

[[phillip course [algebra term spr83]] grade b]

[[tanya course [algebra term fall81]] grade a]

[[greg course [algebra term spr80]] grade f]

[[bliss course [algebra term fall85]] grade d]

[[richard course [algebra term fall82]] grade c]

[[monica course [algebra term spr81]] grade b]

[[howard course [algebra term fall86]] grade a]

[[catherine course [algebra term spr85]] grade f]

[[crissie course [algorithms term spr83]] grade b]

[[ralph course [algorithms term spr84]] grade a]

[[nancy course [algorithms term fall86]] grade f]

[[avery course [algorithms term fall86]] grade d]

[[gifford teaches [algorithms term fall84]] room 12-266]

[[quillan teaches [algorithms term fall83]] room 9-423]

[[kingery teaches [algorithms term fall86]] room 9-423]

[[eager teaches [algorithms term spr83]] room 9-423]

[[king teaches [algorithms term spr85]] room 26-231]

[[caruso teaches [algorithms term spr84]] room 51-114]

[[abelson teaches [algorithms term spr81]] room 4-149]

Figure 3.2. Example of a school data base

and two versions of the extraction functions

pull and p − pull.

The interesting combinations that were tested are:

Comb#	member-fn	equal-fn	pull-fn
0	S	S	S
1	P1	P1	P
2	P2	P2	P
3	P2	P1	P
4	P2	P1	S

Figure 3.3. Parallel Version Combinations

The function **matching-links** is responsible for handling the different combinations of the algorithms mentioned above. Combination 0 was implemented by **matching-links**, 1 by **p-matching-links**, 2 by **p2-matching-links**, and 3 by **p3-matching-links**. Combination 3 uses the first parallel version of the equal function (Figure 2.7) to test the assumption stated earlier that there should not be a significant difference between versions 2 and 3 for **equal-fn** because equal is a quick operation and the overhead of unecessary futures are eliminated.

Combination 4 was derived after conclusion of the tests for combinations 1–3. The **equal-fn** and **member-fn** were taken from the winning combination (combination 3), and joined with the sequential version of the **pull-fn** to measure the effects of parallelism in the pull functions.

3.2. The Programs

3.2.1 The Family Tree Network

The possible queries for a family tree network can pertain to parents, children, sisters, brothers, cousins, and various other family relationships. The ones chosen for testing are cousins and grandchildren.

To find someone's cousins, first their parents are found, then the parents' siblings, then their siblings' children. The query program is shown in Figure 3.4.

```
(defun cousins (child)
       (pull-objects
       (matching-links (pull-objects
                       (matching-links (pull-subjects
                                       (matching-links *wildcard*
                                                       PARENT-OF
                                                       child))
                                       SIBLING-OF
                                       *wildcard* ))
                       PARENT-OF
                       *wildcard* )))
```

Figure 3.4. Cousins Program

This query has the effect of first filtering the *relation* and *subject* on the *OBJECT FALLON* which produces 2 links since *FALLON* has 2 parents. Then the list of links if filtered on the *relation SIBLING-OF* and the *relation PARENT-OF* which means filtering through long property lists.

```
(defun grandchildren (parent)
        (pull-objects (matching-links
                        (pull-objects (matching-links parent
                                                      PARENT-OF
                                                      *wildcard* ))
                                      PARENT-OF
                                      *wildcard* )))
```

Figure 3.5. Grandchildren Program

Finding grandchildren is simply a matter of getting childrens' children, as shown in Figure 3.5.

3.2.2 The School Network

The possible queries for a school network can pertain to various combinations of professors, classes, students, classsrooms, and grades. The queries for testing are finding the professor of a certain class, finding the professors of a student, and obtaining a student's grades in a class.

The program to obtain the professor of a class is shown in Figure 3.6. Figure 3.7 shows the program to find the professors of a given student, and Figure 3.8 shows the program to obtain the grades of a student for a class. The program **student-grades** only has 4 combinations because, since it does not contain any pull functions, combination 4 would be the same as combination 3.

3.3. Measurements

Measurements were taken of the query programs described above: 5 programs with 5 different combinations each, except for the **student-grades** program which had only 4 combinations. These five queries were chosen from all the eligible queries

```
(defun class-professor (class term)
        (pull-subjects (matching-links *wildcard*
                                       TEACHES
                                       (matching-links class TERM term))))
```

Figure 3.6. Class-Professors Programs

```
(defun student-professors (student)
        (pull-subjects
        (matching-links *wildcard*
                        TEACHES
                        (pull-objects (matching-links student
                                                      COURSE
                                                      *wildcard* )))))
```

Figure 3.7. Student-Professors Program

by simulating runs on the VAX and picking the ones that offered the most diverse results. Each query was run with the same inputs at all times, and the outputs ranged from one link to a list of thirteen elements. Figure 3.9 shows the inputs and outputs to the five queries.

CONCERT was used to measure performance times using 1, 2, 3, 4, and 5 processors. Measurements were taken by using the *time* function available on the CONCERT Multilisp. Query times which appeared to have included garbage collection were discarded and re-run. Two to three times were averaged to obtain the numbers which were used to plot the graphs shown in Figures 3.10 to 3.14. The numbers were plotted

```
(defun student-grades (student class term)
        (matching-links (matching-links student
                                        course
                                        (matching-links class TERM term))
                        GRADE
                        *wildcard* )))
```

Figure 3.8. Student-Grades Program

on log-log graphs to show results of speedup effectively. CONCERT using 8 processors was used to measure times for **cousins, p-cousins, p2-cousins, grandchildren, p-grandchildren, p2-grandchildren**, and **p3-grandchildren**. The rest of the numbers for 8 processors were obtained by using the simulations on the VAX. The times obtained from the simulations were usually about 2–3 times faster than the CONCERT results (for queries where the actual CONCERT times were known), so they were scaled up for comparison with CONCERT times by multiplying the simulated time by—the actual time for 1 CONCERT processor divided by the VAX simulated time for 1 processor. This method was checked by comparing the *scaled simulated times* to known CONCERT times, and they were very similar.

The simulated times on the graphs are shown by plotting symbols drawn in dashed lines while the actual CONCERT times are represented by plotting symbols in solid lines.

3.4. Discusssion

Upon examination of the graphs, the sequential and first parallel versions of the programs follow an almost identical path. This indicates that the first parallel version does not make a significant improvement on the retrieval time from the sequential version. The fact that the parallel version is always higher than the sequential can

Query Program I: Cousins

input: "angeli"

output:(corinne hillary linda carol quentin jesse estelle zeke)

Query Program II: Grandchildren

input: "willis"

output:(roscoe toni avery dawn billie lee duke brett althea maureen stanley cary carlos chauncy neville perry trisha fred diego)

Query Program III: Class-Professor

input: (english fall85)

output:(quillan)

Query Program IV: Student-Professors

input: "blake"

output:(quillan fisher rosenweig weitzman saloner carrol reiche yurek halstead beck wnek katz corbato)

Query Program V: Student-Grades

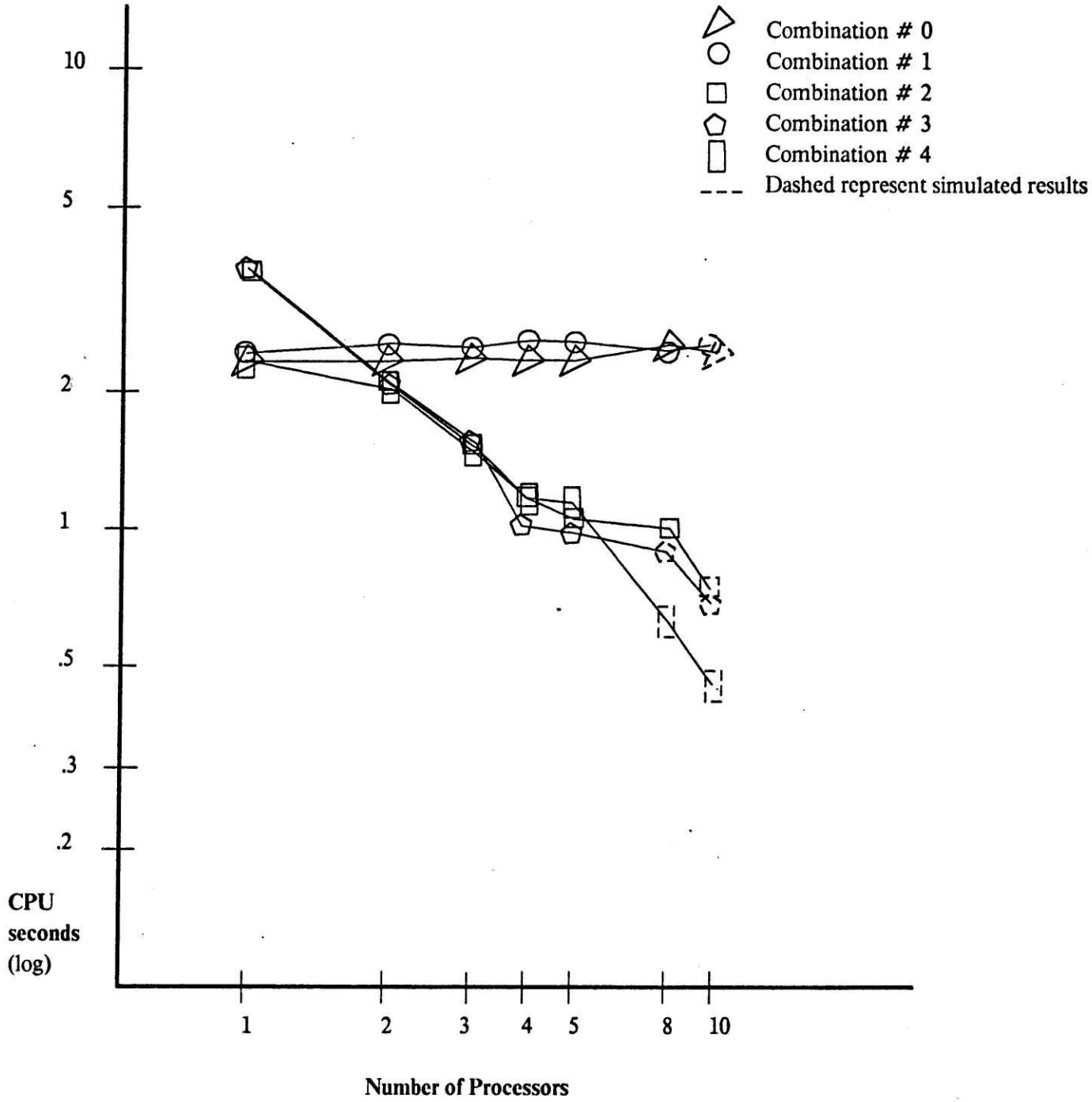
input: (blake computers spr80)

output:[[blake course [computers term spr80]] grade A]

Figure 3.9. Program Summary: Inputs and Outputs

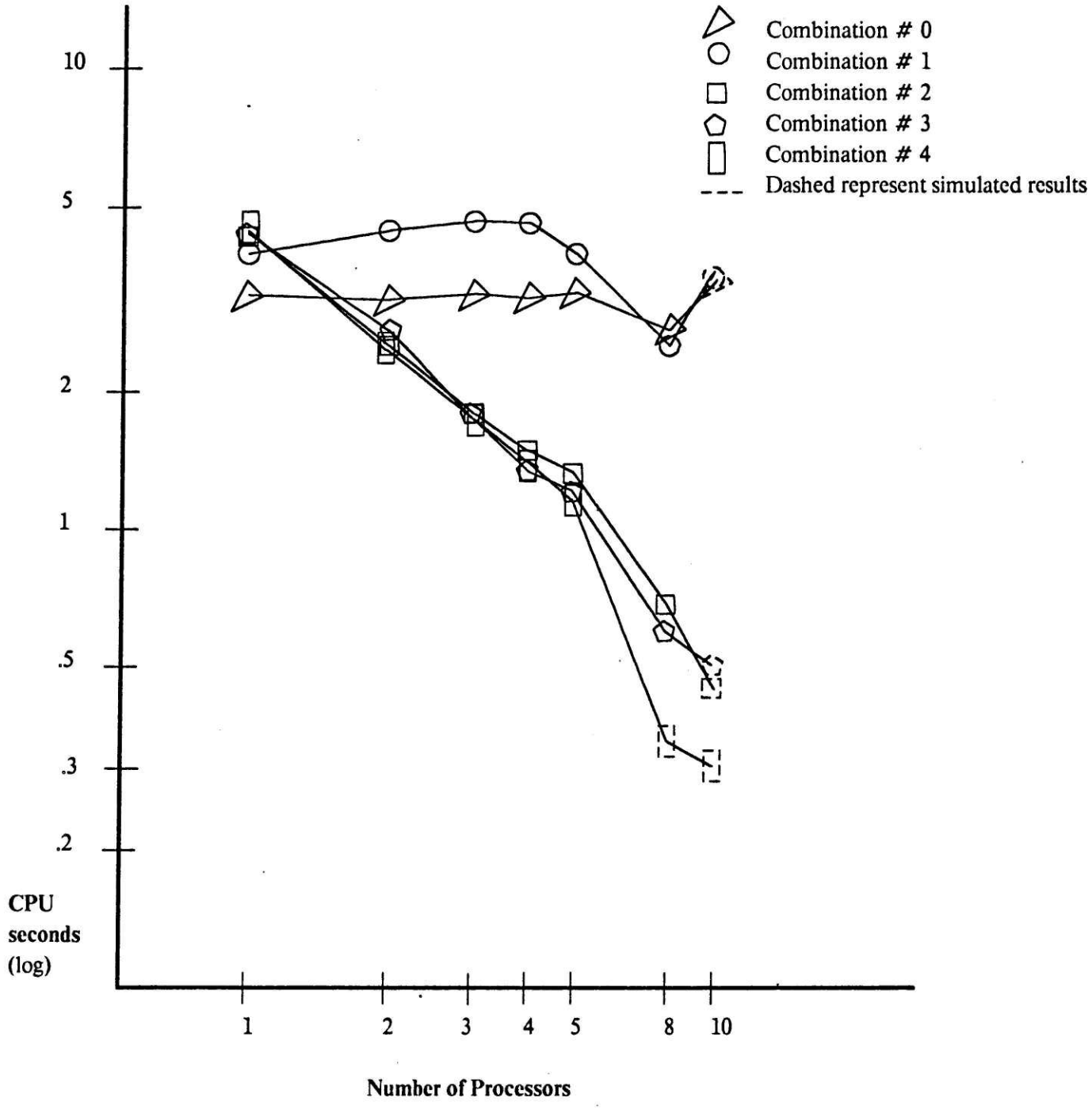
be explained by the overhead associated with the future calls, while providing no real speedup in time.

In all the graphs, the second and third parallel versions produce very similar results, the lines are very close together, even overlapping in some cases. This result is based on the earlier hypothesis that that the two versions of **remove-if-not-equal** are similar because **eq** is a quick operation. In the **class-prof** graph, since the times are much smaller than the rest, the lines are less steep than in the rest of the graphs. This



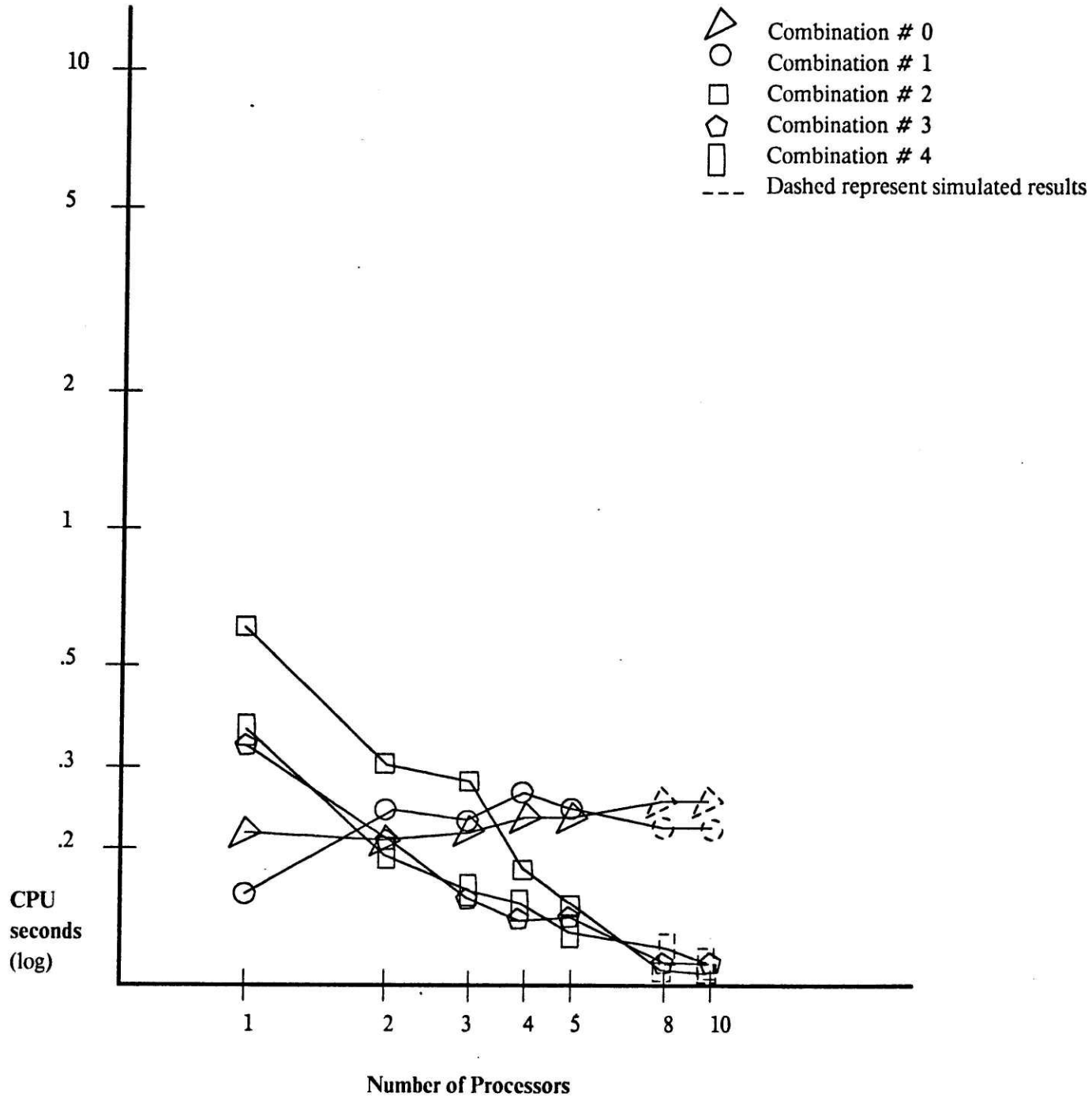
Graph I: Cousins Programs

Figure 3.10



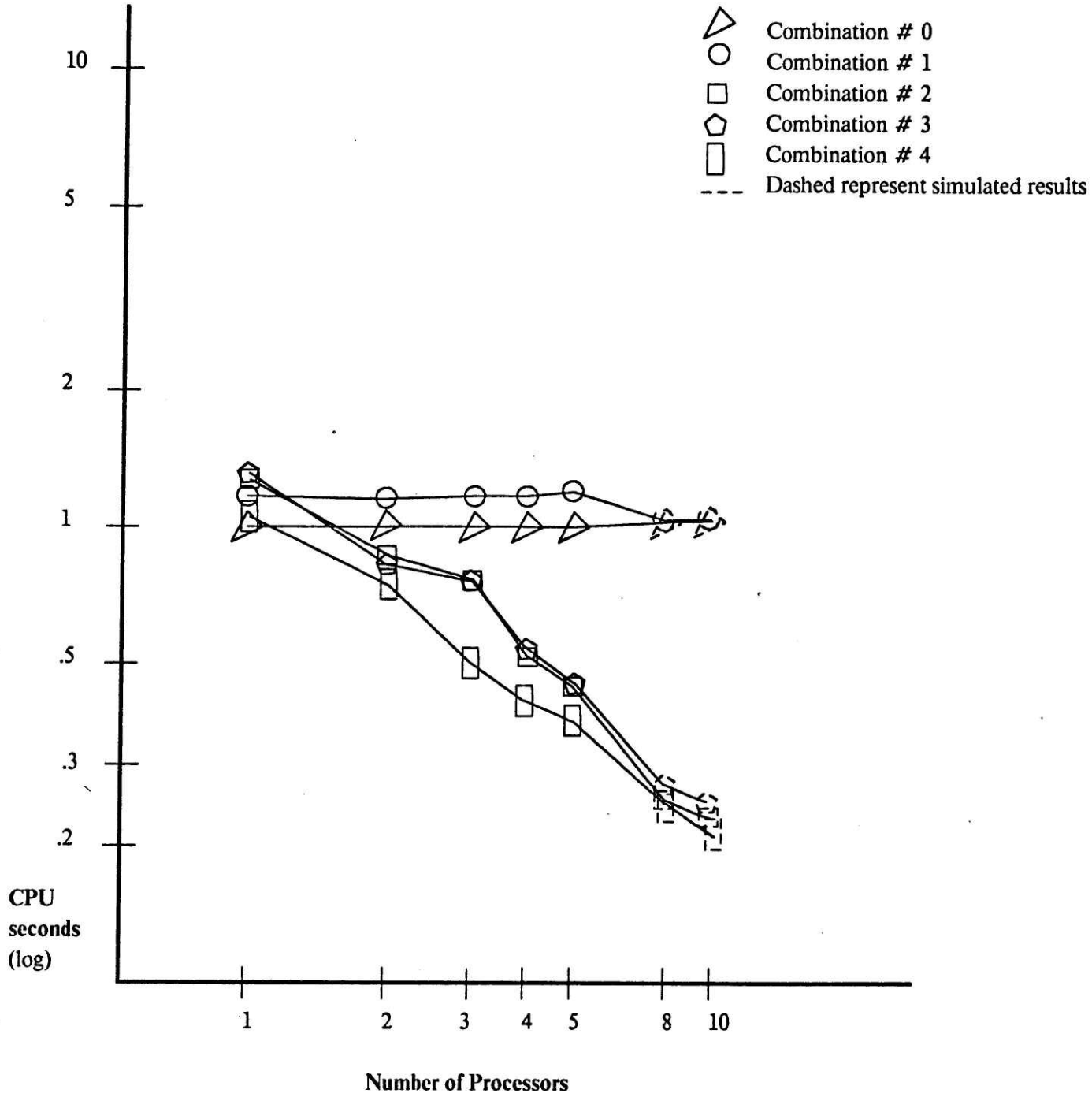
Graph II: Grandchildren Programs

Figure 3.11



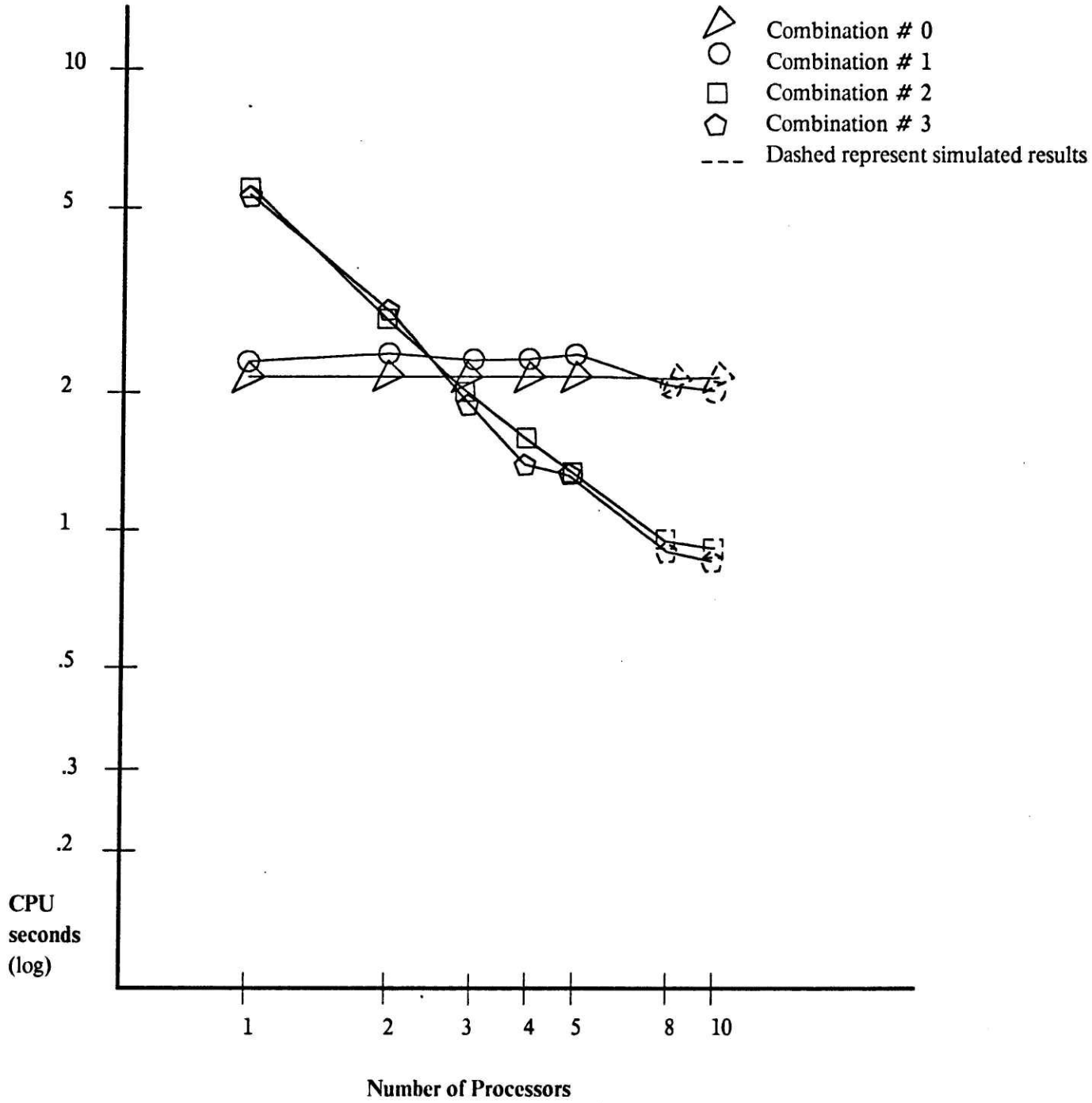
Graph III: Class-Professors Programs

Figure 3.12



Graph IV: Student-Professors Programs

Figure 3.13



Graph V: Student-Grades Programs

Figure 3.14

can happen because each time **matching-links** is called in **class-prof** , it produces only one link, therefore the opportunities for parallelism are not as great as in the other queries which often produce a list of many links as an outcome of a call to **matching-links** .

In the cases where the *best* programs were tested, the results were almost identical to the corresponding **p-3** parallel versions. This may be explained by the fact that the largest answers consisted of lists of 19 elements, not enough to exploit the parallelism available in the **pull-subjects** and **pull-objects** functions.

In **graph 1** and **graph 2** , the best parallel versions, which are alterations of the combination 3 functions in both cases, the lines fall faster as the number of processors increase than in the other graphs. This is surprising since **cousins** contains three instances of **pull-objects** and **grandchildren** contains two instances of **pull-objects**. This might indicate that in cases where the pull functions and **matching-links** alternate, the added overhead of the futures in the pull functions might not be worth it.

The size of the networks does not seem to make a difference in the ability of the parallel algorithms, since there are no striking differences between graphs 1 and 2 and graphs 3, 4, and 5.

Chapter IV

Conclusion

The results of this experiment show a future for parallelism in the retrieval algorithms applied to semantic networks. The tests proved more successful for queries that must manipulate a lot of data than for queries that did not have to process as much information, as shown by the graphs in the last chapter.

Although the results conclude no significant difference in speedups due to the difference in size of the two networks, the networks used were not as large as most useful semantic nets would be. This was due to the constaints of space and time from working on the VAX. A truly useful knowledge-based system would be much too large to use on an experimental scale. Given that parallelism is more useful when there are more opportunities for it, retrieval in the larger networks should be enchanced significantly. Given more time, this hypothesis could be a possible extension of this research.

Appendix A

Source Code Listings

```
;;; -*- Mode: LISP; Syntax: Common-lisp; Package: MULTILISP; -*-
;;;;This defines database structures and their access functions.

;;;Free variables.

(setf assertion-window t)

(defvar *wildcard* '*wldcard*)
(defvar *things* nil)
(defvar *relations* nil)
(defvar *link-count* 0)
(defvar *time* 0)

;; Macros to access various slots of the link array.

(defmacro link-subject (l) '(aref ,l 0))
(defmacro link-relation (l) '(aref ,l 1))
(defmacro link-object (l) '(aref ,l 2))

(defmacro link-subject-of (l) '(aref ,l 3))
(defmacro link-object-of (l)  '(aref ,l 4))
(defmacro link-truth-value (l) '(aref ,l 5))
(defmacro link-time (l) '(aref ,l 6))
(defmacro relation-of (r) '(get ,r 'relation-of))

(defun link-subject-fn (l)  (link-subject l))
(defun link-relation-fn (l) (link-relation l))
(defun link-object-fn (l) (link-object l))

;; Rewrite car and  cdr to run on the VAX Multilisp.

(defun my-car (l)
  (cond ((consp l) (car l))
        (t nil)))

(defun my-cdr (l)
  (cond ((consp l) (cdr l))
        (t nil)))

;; Function to create a link given the subject, relation, and object.
;; Truth value is unknown if not given.
;; Subject-of and object-of lists start out nil.

(defun make-link (subj rel obj &optional (truth 'unknown))
 (let ((m (make-array 7)))
  (setf (link-subject m) subj)
  (setf (link-relation m) rel)
  (setf (link-object  m) obj)
  (setf (link-truth-value m) truth)
  (setf (link-subject-of m) nil)
  (setf (link-object-of m) nil)
  m))

;;  Functions in common lisp not existing in MLISP defined here

;; endp: returns true if argument is null.
(defmacro endp (n) '(null ,n))

;; link-p: returns true if the arguement is a link (array).
(defmacro link-p (n) '(arrayp ,n))

;; listp: returns true if the argument is a list.
(defun listp (n)
  (cond ((null n) t)
        ((not (atom n)) t)
        (t nil)))

;; rest: returns the cdr of the argument l which is expected to be a list.
(defmacro rest (l) '(cdr ,l))

;; adjoin: returns a list that is the union of the two arguments: item and list
(defun adjoin (item list)
  (if (member item list) list (cons item list)))
```

```
;; unless: first evaluates p, if the result is not nil, then a is not  evaluated
;; and nil is returned. Otherwise, a is evaluated and the result of a is returned.
(defmacro unless (p a)
  `(cond ((not ,p) ,a)
         (t nil)))

;; when: first evaluates p, if the result is nil, then a is not evaluated,
;; and nil is returned. Otherwise, a is evaluated and the result of a is returned.
(defmacro when (p a)
  `(cond (,p ,a)
         (t nil)))

;; when2: first evaluates p, if the result is nil, then a and b are not evaluated,
;; and nil is returned. Otherwise, a and b are evaluated in order, and the
;; result of b is returned.
(defmacro when2 (p a b)
  `(cond (,p ,a ,b)
         (t nil)))



;; Format-links: takes as argument a list of links
;; and prints each link of the list.
(defun format-links ( story)
     (do ((l (car story) (car story))
          (story (cdr story) (cdr story)))
         ((null l))
         (format-link  l)))

;; Format-link: prints a line feed, and then the link.
(defun format-link (link)
   (princ "
")
   (bracket link)
   t)

;; Bracket: prints a  link as a string with brackets delimiting the link.
(defun bracket (link)
  (cond ((link-p link)
         (princ "[ ")
         (bracket (link-subject link))
         (princ " ")
         (princ (get-truth link))
         (bracket (link-relation link))
         (princ " ")
         (bracket (link-object link))
         (princ " ] "))
        ((null link))
        (t (princ link))))


;; Get-truth: returns the printing character equivalent of the truth
;; value of the  argument link.
;;    Returns: + if true
;;             - if false
;;             ? if unknown
(defun get-truth (link)
  (let ((truth-value (link-truth-value link)))
    (cond ((equal truth-value 'true) '+)
          ((equal truth-value 'false) '-)
          (t '?))))


;; Make-object: if the argument object is not already an object in
;; the network, it makes it one, and places it in the universal list of things.
(defun make-object (object)
  (unless (subject-of object)
    (push object *things*))
  object)
```

```
;;;This is where assertion making is done.  General strategy is to find
;;;all links with appropriate subject, link, and object,
;;;then find the most recent or make a new link, if none
;;; if same, do nothing
(defun get-link (s r o &optional (links (subject-of s)))
  (if links
      (let ((link (car links)))
        (cond ((and (eq s (link-subject link))
                    (eq r (link-relation link))
                    (eq o (link-object link)))
               (princ "*")
               link)
              (t (get-link s r o (cdr links)))))))


;; Build-link: takes as arguments the subject, relation and object, and
;; optional truth value, and creates a link for them.
;; Also establishes the subject-of and object-of lists when
;; the subjects or objects are links themselves.
(defun build-link (s r o &optional (truth-value 'unknown))
  (let ((link (make-link s r o   truth-value)))
    (unless (link-p s) (record-object s))
    (unless (link-p o) (record-object o))
    ;;
    ;; ADDITION HERE:
    ;;
    (SETF (GET R 'RELATION-OF) (PUSH LINK (GET R 'RELATION-OF)))
    ;;
    (if (link-p s)
        (setf (link-subject-of s) (push link (link-subject-of s)))
        (setf (get s 'subject-of) (push link (get s 'subject-of))))
    (cond ((link-p o)
           (setf (link-object-of o)
                 (push link (link-object-of o))))
          ((forbidden-thing? o)
           NIL)
          (T
           (setf (get o 'object-of)
                 (push link (get o 'object-of)))))
    (setf *link-count* (add1 *link-count* ))
    (princ "+")
    LINK))

;; Record-relation: keeps track of relation atoms by adding them to the list of relations.
(defun record-relation (r)
  (unless (member r *relations*)
          (push r *relations*)))

;; Record-object: keeps track of objects by adding them to the list of objects.
(defun record-object (o)
  (unless (or (subject-of o) (object-of o))
    (push o *things*)))



;; My-assert: this function is used to assert the truth-value of a link.
(defun my-assert (s r o new-truth-value &aux old-truth-value)
  (let ((link (get-link s r o)))
    (when link (setf old-truth-value (link-truth-value link)))
    (cond ((not link)
           (setf link (build-link s r o new-truth-value)))
          ((eq new-truth-value old-truth-value))
          ((eq 'unknown old-truth-value)
           (setf (link-truth-value link) new-truth-value)
           (setf (link-time link) *time*))
          (t (setf link (build-link s r o new-truth-value))))
    link))


(defun affirm (s r o) (my-assert s r o 'true) )

(defun deny (s r o) (my-assert s r o 'false) )

(defun question (s r o) (my-assert s r o 'unknown) )

(defun consider (s r o) (my-assert s r o 'unknown) )
```

```
;; Subject-of: Extracts the list of links that thing is a subject of.
(defun subject-of (thing)
  (cond ((link-p thing)
         (link-subject-of thing))
        (T
         (my-get thing 'subject-of))))

;; Object-of: Extracts the list of links that thing is an object of.
(defun object-of (thing)
  (cond ((link-p thing)
         (link-object-of thing))
        (T
         (my-get thing 'object-of))))


;; Pull-Subjects: Takes a list of links and  returns a list of all the subjects of the
;; links in the list.
(defun pull-subjects (links)
  (if (null links)
      nil
      (cons (link-subject (car links))
            (pull-subjects (cdr links)))))



;; Pull-Objects: Takes a list of links and returns a list of all the objects of the
;; links in the list.
(defun pull-objects (links)
  (if (null links)
      nil
      (cons (link-object (car links))
            (pull-objects (cdr links)))))




;;; Matching functions

(defun remove-if-not-equal (filter  links filter-function)
  (cond ((null links) nil)
        ((eq (filter-function (car links)) filter)
         (cons (car links) (remove-if-not-equal filter (cdr links) filter-function)))
        (t (remove-if-not-equal filter (cdr links)  filter-function))))

(defun remove-if-not-member (filter links filter-function)
  (cond ((null links) nil)
        ((member (filter-function (car links)) filter)
         (cons (car links) (remove-if-not-member filter (cdr links) filter-function)))
        (t (remove-if-not-member filter (cdr links) filter-function))))



(defun filter-links (filter links filter-function)
  ;; EFFECTS: space saver, filter-slot is  'subject
  (cond ((eq filter *wildcard*) links)
        ((or (atom filter) (link-p filter))
         (remove-if-not-equal filter links filter-function))
        ((listp filter)
         (remove-if-not-member filter links filter-function))))


;; This macro returns true if the 'thing' is not a wildcard and not nil.
(defmacro not-wild (thing)
  `(and ,thing (not (eq ,thing *wildcard*))))



;; Function to format print the links.
(defun show-matching-links ( subject  relation object)
 (let ((links (matching-links  subject relation object)))
  (format-links  links)
  links))
```

```
(defun matching-links (subject  relation object)
  ;;
  ;; REQUIRES: At least one of the args must be atomic
  ;;
  ;; EFFECTS: Will take subject object and relations that are either
  ;; atoms or nil or predicates or a list of atoms or a list of links.
  ;; If nil then that slot in a link will
  ;; be considered a wild card and all fillers will satisfy a match there.
  ;; If a predicate is there than all fillers which satisfy the predicate
  ;; will match.  If an atom then only fillers which are EQ will match.
  ;;
  ;; RETURNS: Matching links in a list.
  ;;
  (cond ((and  (not-wild subject)  (or (atom subject) (link-p subject)))
         (filter-links object
                       (filter-links relation (subject-of subject) link-relation-fn)
                       link-object-fn))
        ;;
        ;; If RELATION was null then just leave mlinks unaltered since null
        ;; matches all relations.
        ;;
        ((and  (not-wild object) (or (atom object) (link-p object)))
         (filter-links subject
                       (filter-links relation (object-of object) link-relation-fn)
                       link-subject-fn))
        ((and  (not-wild relation) (or (atom relation) (link-p relation)))
         ;;
         ;; relation is the only thing specified as atom.
         ;;
         (filter-links object
                       (filter-links subject (relation-of relation) link-subject-fn)
                       link-object-fn))
        (T (error "At least one of the match arguments must be atomic"))))



(defun convert-query (qlist &optional (dont-show nil))

 ;;;
 ;;; EFFECTS: Will convert a list like:
 ;;; (GET-LINK JOHN WANT (GET-LINK JOHN EAT WH)) to:
 ;;; (matching-links :subject 'john :relation 'want
 ;;;                 :object (matching-links :subject 'john :relation 'eat))
 ;;; If dont-show is nil than show the links formatted.  If T then don't
 ;;; show them formatted.  This prevents showing nested queries.
 ;;;
 ;;; RETURNS: A list of the links satisfying the query.
 ;;;
 (if (eq (car qlist) 'get-link)
    (let ((query-list nil)
          (subject (second qlist))
          (relation (third qlist))
          (object (fourth qlist)))
      (cond ((and (not (atom subject)) (eq (car subject) 'get-link))
             (push (convert-query subject T) query-list))
            ((not (or (eq subject 'wh)
                      (eq subject 'who)))
            (push subject query-list)))
      (cond ((and (not (atom relation)) (eq (car relation) 'get-link))
             (push (convert-query relation T) query-list))
            ((not (or (eq relation 'wh)
                      (eq relation 'who)))
            (push relation query-list)))
      (cond ((and (not (atom object)) (eq (car object) 'get-link))
             (push (convert-query object T) query-list))
            ((not (or (eq object 'wh)
                      (eq object 'who)))
            (push object query-list)))
      (if dont-show
         (apply #'matching-links (reverse query-list))
         (apply #'show-matching-links (reverse query-list))))
    (error "Improper argument to Convert-query")))
```

```
(defun tell (q-list)
 (convert-query q-list))



;; Statistics: tells how many objects and links in the network.
(defun statistics ()
  (princ "
There are ")
  (princ (length *things*))
  (princ " objects and ")
  (princ *link-count*)
  (princ " links")
  t)


;;;This is for rapid link creation.


(defmacro r (&rest rest) (list 'record-aux 'rest))

(defmacro a (&rest rest) (list 'record-aux '(cons 'a rest)))

(defmacro d (&rest rest) (list 'record-aux '(cons 'd rest)))

(defmacro q (&rest rest) (list 'record-aux '(cons 'q rest)))


(defun record-aux (s)
  (cond ((null s) nil)
        ((atom s) '',s)
        ((endp (rest s)) (record-aux (first s)))
        ((= 3 (length s)) (cons 'affirm (mapcar #'record-aux s)))
        (t (cons (let ((tag (first s)))
                   (cond ((equal tag 'a)  'affirm)
                         ((equal tag 'd)  'deny)
                         ((equal tag 'q)  'question)))
                 (mapcar #'record-aux (rest s))))))


(defun forbidden-thing? (thing)
  (or (numberp thing)
      (listp thing)))

(defun my-get (item prop)
  (cond ((forbidden-thing? item)
         NIL)
        (T (get item prop))))
```

```
;;; -*- Package: MULTILISP; Syntax: Common-lisp -*-
;;; This file contains the parallel retrieval algorithms.



;; COMBINATION # 1
;; remove-if-not-member: P1
;; remove-if-not-equal : P1
;; pull-xxx            : P


;; P-filter-links:
;;   arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      If filter is a wildcard then nothing is done, else
;;      the list of links are searched
;;      for links with elements matching filter.
;;      This version calls future of the matching function.
(defun p-filter-links (filter links filter-function)
  (cond ((eq filter *wildcard*))
        ((or (atom filter) (link-p filter))
         (p-remove-if-not-equal filter links filter-function))
        ((listp filter)
         (p-remove-if-not-member filter links filter-function))))



;; P-remove-if-not-equal
;; arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      given the list of links, removes all the links whose filter-function
;;      slot values are not equal to filter.
;; This version calls a future of the recursive call to itself.
(defun p-remove-if-not-equal (filter links filter-function)
  (cond ((null links) nil)
        ((eq (filter-function (my-car links)) filter)
         (cons (my-car links)
               (future (p-remove-if-not-equal filter (my-cdr links) filter-function))))
        (t (p-remove-if-not-equal filter (my-cdr links) filter-function))))



;; P-remove-if-not-member
;; arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      given the list of links, removes all the links whose filter-function
;;      slot values are not members of the list  filter.
;; This version calls a future of the recursive call to itself.
(defun p-remove-if-not-member (filter links filter-function)
  (cond ((null links) nil)
        ((member (filter-function (my-car links)) filter)
         (cons (my-car links)
               (future (p-remove-if-not-member filter (my-cdr links) filter-function))))
        (t (p-remove-if-not-member filter (my-cdr links) filter-function))))
```

```
;; P-matching-links
;; arguments:
;;    subject   - the subject of the link
;;    relation  - the relation of the link
;;    object    - the object of the link
;; effects:
;;    first checks the subject, relation, then object,
;;    obtains a list of links to start with, and filters
;;    them matching on the other two slots.
;;    This version  calls p-filter-links.
(defun p-matching-links (subject relation object)
  (cond ((and (not-wild subject) (or (atom subject) (link-p subject)))
         (p-filter-links object
                        (p-filter-links relation (subject-of subject) link-relation-fn)
                        link-object-fn))
        ;;
        ;; If RELATION was null then just leave mlinks unaltered since null
        ;; matches all relations.
        ;;
        ((and  (not-wild object) (or (atom object) (link-p object)))
         (p-filter-links subject
                        (p-filter-links relation (object-of object) link-relation-fn)
                        link-subject-fn))
        ((and  (not-wild relation) (or (atom relation) (link-p relation)))
         ;;
         ;; relation is the only thing specified as atom.
         ;;
         (p-filter-links object
                        (p-filter-links subject (relation-of relation) link-subject-fn)
                        link-object-fn))
        (T (error "At least one of the match arguments must be atomic"))))
```

```
;; COMBINATION # 2
;; remove-if-not-member: P2
;; remove-if-not-equal : P2
;; pull-xxx          : P
```

```
;; P2-remove-if-not-equal
;; arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      given the list of links, removes all the links whose filter-function
;;      slot values are not equal to filter.
;; This version calls a future of the recursive call to itself.
(defun p2-remove-if-not-equal (filter links filter-function)
  (if (null links)
      nil
      (let ((rest (future (p2-remove-if-not-equal filter (my-cdr links) filter-function))))
        (if (eq (filter-function (my-car links)) filter)
            (cons (my-car links) rest)
            rest))))
```

```
;; P2-remove-if-not-member
;; arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      given the list of links, removes all the links whose filter-function
;;      slot values are not members of the list  filter.
;; This version calls a future of the recursive call to itself.
(defun p2-remove-if-not-member (filter links filter-function)
  (if (null links)
      nil
      (let ((rest (future (p2-remove-if-not-member filter (my-cdr links) filter-function))))
        (if (member (filter-function (my-car links)) filter)
            (cons (my-car links) rest)
            rest))))
```

```
;; P2-filter-links:
;;   arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      If filter is a wildcard then nothing is done, else
;;      the list of links are searched
;;      for links with elements matching filter.
;;      This version calls future of the matching function.
(defun p2-filter-links (filter links filter-function)
  (cond ((eq filter *wildcard*))
        ((or (atom filter) (link-p filter))
         (p2-remove-if-not-equal filter links filter-function))
        ((listp filter)
         (p2-remove-if-not-member filter links filter-function))))
```

```
;; P2-matching-links
;; arguments:
;;    subject   - the subject of the link
;;    relation  - the relation of the link
;;    object    - the object of the link
;; effects:
;;    first checks the subject, relation, then object,
;;    obtains a list of links to start with, and filters
;;    them matching on the other two slots.
;;    This version  calls p2-filter-links.
(defun p2-matching-links (subject relation object)
  (cond ((and (not-wild subject) (or (atom subject) (link-p subject)))
         (p2-filter-links object
                         (p2-filter-links relation (subject-of subject) link-relation-fn)
                         link-object-fn))
        ;;
        ;; If RELATION was null then just leave mlinks unaltered since null
        ;; matches all relations.
        ;;
        ((and  (not-wild object) (or (atom object) (link-p object)))
         (p2-filter-links subject
                         (p2-filter-links relation (object-of object) link-relation-fn)
                         link-subject-fn))
        ((and  (not-wild relation) (or (atom relation) (link-p relation)))
         ;;
         ;; relation is the only thing specified as atom.
         ;;
         (p2-filter-links object
                         (p2-filter-links subject (relation-of relation) link-subject-fn)
                         link-object-fn))
        (T (error "At least one of the match arguments must be atomic"))))
```

```
;; COMBINATION # 3
;; remove-if-not-member: P2
;; remove-if-not-equal : P1
;; pull-xxx          : P


;; P3-filter-links:
;;   arguments:
;;      filter  - the object to search for
;;      links   - the list of links to be searched
;;      filter-function  - the function to extract the correct slot of the link
;;   effects:
;;      If filter is a wildcard then nothing is done, else
;;      the list of links are searched
;;      for links with elements matching filter.
;;      This version calls future of the matching function.
(defun p3-filter-links (filter links filter-function)
  (cond ((eq filter *wildcard*))
        ((or (atom filter) (link-p filter))
         (p-remove-if-not-equal filter links filter-function))
        ((listp filter)
         (p2-remove-if-not-member filter links filter-function))))




;; P3-matching-links
;; arguments:
;;    subject   - the subject of the link
;;    relation  - the relation of the link
;;    object    - the object of the link
;; effects:
;;    first checks the subject, relation, then object,
;;    obtains a list of links to start with, and filters
;;    them matching on the other two slots.
;;    This version  calls p3-filter-links.
(defun p3-matching-links (subject relation object)
  (cond ((and (not-wild subject) (or (atom subject) (link-p subject)))
         (p3-filter-links object
                         (p3-filter-links relation (subject-of subject) link-relation-fn)
                         link-object-fn))
        ;;
        ;; If RELATION was null then just leave mlinks unaltered since null
        ;; matches all relations.
        ;;
        ((and  (not-wild object) (or (atom object) (link-p object)))
         (p3-filter-links subject
                         (p3-filter-links relation (object-of object) link-relation-fn)
                         link-subject-fn))
       ((and  (not-wild relation) (or (atom relation) (link-p relation)))
         ;;
         ;; relation is the only thing specified as atom.
         ;;
         (p3-filter-links object
                         (p3-filter-links subject (relation-of relation) link-subject-fn)
                         link-object-fn))
       (T (error "At least one of the match arguments must be atomic"))))
```

```
;; PARALELL PULL FUNCTIONS

;; P-pull-subjects
;; arguments:
;;    links  - the list of links of which to obtain the subject elements
;; effects:
;;    returns a list of the subjects of each link  in the list of links.
;; This version calls the future of the recursive call to itself.
(defun p-pull-subjects (links)
  (if (null links)
      nil
      (my-cons (link-subject (my-car links))
            (future (p-pull-subjects (my-cdr links))))))

;; P-pull-objects
;; arguments:
;;    links  - the list of links of which to obtain the object elements
;; effects:
;;    returns a list of the objects of each link  in the list of links.
;; This version calls the future of the recursive call to itself.
(defun p-pull-objects (links)
  (if (null links)
      nil
      (my-cons (link-object (my-car links))
            (future (p-pull-objects (my-cdr links))))))


(defmacro p-get-subjects (object &optional (relation *wildcard*))
  '(pull-subjects (p-matching-links *wildcard* ,relation ,object)))

(defmacro p-get-objects (subject &optional (relation *wildcard*))
  '(pull-objects (p-matching-links ,subject ,relation *wildcard*)))
```

```
;;; -*- Package: MULTILISP; Syntax: Common-lisp -*-

;; The list of male names.
(defvar  lott '(blake jane alexis fallon michael joan crystal steven jeff claudia
                andrew phillip sable cary yona richard cecil monica bliss
                francesca amanda dominique ken howard carla cherly catherine
                anthony daniel david greg patrick jennifer donald gene
                robert henry james ted cora cory paul eric))

(defvar m-names '(blake michael eric james jeff ronald donald paul gene ted robert
                  anthony tony ken howard richard greg phillip danny henry andrew
                  gary sam kevin thomas  fred dennis brett  harry eddie
                  jack charles frank rick max harrison leo gerald george
                  murray  arthur avery ralph scott boris martin allen brian raymond
                  oren timothy timmy craig peter bobby oliver alberto leonard harvey
                  hank eugene lowell franklin walter benjamin douglas brad nathan
                  hershel harold bart alexander bartholomew adam cain abel victor
                  anotonio vincent burt ernie bruce felix oscar matthew lincoln
                  othello hamlet vladimir diego bjorn mats mikhail jamal gavin
                  archie archibald guillermo roman vadim cyrano nathaniel
                  everett efrem elijah sidney
                  dane zeke zachary ross rusty lee broderick rodrigo rocky
                  abraham noah lenny tyberius salvador caesar chuck roland raoul
                  aaron fabian hugh herbert hughbert ozzie randolph irving
                  irvin ben rodney gordon morris melvin duncan dunbar bernie
                  myron byron brant earl eliot isaac stanley alessandro ricardo
                  monty russel nicholas carter barney basil benedict boyd
                  bruno calvin carlyle chauncy claude clayton clifford cornielius
                  cyrus demetrius derrick drew dwight ebenezer enrico
                  perry terry tony jim dick manny carlos laurence larry
                  charleton roger  christopher william billy marty dominic
                  randy harley jonathan don
                  alfred dean lucas karl kirk kyle jesse hal conrad ramsey todd van
                  cary joshua luke mark john  cecil patrick joseph sean
                  rudolph stuart quincy pierce sumner wyatt winston willis winfred
                  willard  rupert raphael samson theodore kent clark bertrand
                  everett ethan elijah chester  chauncy burl justin barabas lancelot
                  ian immanuel ivan humphrey isaiah  lance lester homer horace herman
                  dirk duke dorian darius ernest elbert omar  neville norman norton
                  moss  rollo rolph roscoe roy tristan noam neil reuben
                  reynold roderick rufus  rutherford samuel vernon vaughn ulysses
                  terence terrel thaddeus thurston tobias  ward waldo vergil
                  wesley warren wilfred wayne rex octavius obadiah nat newton
                  murdoch quentin ludwig huntington lionel geoffrey ezra  ichabod
                  ignatius troy marmaduke johann sebastian franz sayan cabot
                  spike fonzie clay curtis jock tug marshall marcel dale bud
                  elroy cleon cleotis antony popeye dexter wally arty grant
                  apollo mercury zeus poseiden socrates plato spartacus spock
                  anton anatolie horatio hannibal hodge ivor jarvis  jason jed ))


;; The list of female names.
(defvar f-names '(alexis fallon claudia crystal amanda yona sable francesca laura
                  jennifer cherly carla cora catherine monica bliss tanya
                  christine susan melissa mary leslie robin mona teresa
                  paulette toni frances fanny greta ingrid charlotte
                  sherri carrie loni sue mary-ann donna millicent annabelle
                  kim nancy crissie olga inga maria pattie lori lauren
                  michelle ellen betty karen jill ann polly helen linda joanne
                  estelle ethel edith marion ruth rose allison alice
                  sharon joan sandra debbie lola tara grace  ali
                  gretchen matilda madeline  trisha galia jackie suzanne sandy rosanne
                  roxanne bambi angeli candi thespina lissa lisa liza wendy bea
                  carolyn cindy stephanie kamala carol jessica alyssa krisztina holly
                  samantha bertha heidi lorna sheree gwinneth cecelia eva ava rickie
                  suzzette martha marta angelica amy penelope lynn aprohidite
                  venus desiree sherry quinn trudy lolita stacy tracy shirley shiela
                  dara sara melinda nicoletta nadine marchia janice emily rhoda valerie
                  joy deborah phyllis bess gloria jeannie mildred florence tanya judith
                  rhonda brenda brandi mandy mindy  paula pauline jean jamie
                  rebecca julia rachel leah diane diana dina esther marie georgette
                  lois louise rosalyn louella myrna monique gwendelyn bobbie leana
                  leona clarise claire charlene marlene arlene ursala natashia
                  june violet viola petunia petula donna doris priscilla ailish colleen
                  collette dorothy carmen molly vivienne tess anastasia greer eleanor
                  helena alena alona cissy kelly dionne zena myrtle constance katrina
                  hillary agnes agatha wilma veronica farla dana nickie darla carling
```

```
                    darlene martina hannah gabriela billie sylvia estelle valencia ronnie
                    wanda althea rita vera megan maureen maura maren kay farrah jacqueline
                    shelley tiffany pearl opal ruby margaret maggie peggy madge marge
                    lila lillian lilliette olive florence flora  dawn fern avril amelia
                    nellie frieda sally janet tina albertina harriet bunita buffy aretha
                    celeste didi mimi gigi dolly mame glenda patsy linda miriam marilyn
                    audrey anita hazel ida nora noreen kimberly corinne nadia eunice
                    julianne virginia andrea  mickie maude risa nina mina alexandria
                    anabelle appolonia vanity charity hope cher liz trixie coco lydia
                    cleopatra tammy temi demi ally becky blair whitney belinda blanche
                    gail shira sheena marca marisa shann barbara shannon sonny mabel
                    lotte dotty tuesday whilhemina eve lucy pamela))

;; FILE I/O ROUTINES


;; Free variables
(defvar dynasty-filename "vx:/usr/hjb/dynasty.links")
(defvar dynasty nil)

;; This function writes a link onto a file (f) in parenthesis.
(defun write-link (link  f)
  (princ " ( " f)
  (let ((subject (link-subject link)))
    (cond ((link-p subject)
           (write-link subject f))
          (t (princ subject f))))
  (princ " " f)
  (let ((relation (link-relation link)))
    (princ relation f))
  (princ " " f)
  (let ((object (link-object link)))
    (cond ((link-p object)
           (write-link object f))
          (t (princ object f))))
  (princ " ) " f))


;; This function prints a carriages return then writes the link in the file (f).
(defun write-affirm (link f)
  (princ "
" f)
  (write-link link f)
  link)

;; Assert-link
;; given a list representing a list (link), this function
;; makes the list into a real link.
(defun assert-link (link &aux subject relation object)
  (setq subject (my-car link))
  (setq relation (cadr link))
  (setq object (caddr link))
  (cond ((atom subject))
        ((listp subject)  ;; affirm links
         (setq subject (assert-link subject))))
  (cond ((atom object))
        ((listp object)
         (setq object (assert-link object))))
  (affirm subject relation object))

;; Read-file
;; This function reads in a file (given fname) and
;; assumes the object is a list of lists (representing links)
;; and makes links out of each one.
(defun read-file (fname)
  (let ((f (infile fname)))
    (let ((ll (read f)))
      (close f)
      (mapcar assert-link ll))
    t))


;; END OF FILE I/O ROUTINES
```

```
;; Count
;; This function returns the number of elements in l.
(defun count (l)
  (cond ((null l) 0)
        (t (+ 1 (count (my-cdr l))))))


;; Get-rand
;; This functions generates a random number from 0 to n.
(defmacro get-rand (n)
  '(% (rand) ,n))


;; Getn
;; This macro obtains the index-th element of array l.
(defmacro getn (l index)
  '(aref ,l ,index))


;; Setn
;; This macro sets the index-th element of array l to be value.
(defmacro setn (l index value)
  '(setf (aref ,l ,index) , value))


(defmacro when5 (a b c d e f)
  '(cond (,a ,b ,c ,d ,e ,f)))


;; This macro results in true if the child is an orphan and has no
;; children of its own.
(defmacro homeless (child father mother)
  '(and (not (eq ,child ,father))
        (not (eq ,child ,mother))
        (null (get-subjects ,child 'parent-of))))


;; This macro results in true if a and b are not siblings or cousins.
(defmacro not-closely-related (a b)
  '(and (not (member ,b (siblings ,a)))
        (not (member ,b (cousins ,a)))))


(defmacro make-male (m)
  '(write-affirm (affirm ,m 'sex 'male) dynasty))

(defmacro make-female (f)
  '(write-affirm (affirm ,f 'sex 'female) dynasty))


;; Starter
;; This function takes a list and returns an array containing the same elements.
(defun starter ( names &optional (n (length  names)))
  (let ((mm (make-array n)))
    (do ((i 0 (add1 i)))
        ((eq i n))
        (setf (aref mm i) (my-car names))
        (setq names (my-cdr names)))
    mm))


(defmacro evenp (n)
  '(eq (% ,n 2) 0))
```

```
;; This function randomly generates a person.
(defun get-child (n mnames fnames  &aux person)
  (cond ((evenp n)
         (setq person (getn fnames n))
         (make-female person))
        (t
          (setq person (getn mnames n))
          (make-male person)))
  person)



;; This function randomly generates a family using the relations
;; parent-of, sibling-of, and sex.
(defun family (p mn fn &optional (tt p) &aux father mother child)
  (setq dynasty (outfile dynasty-filename))
  (princ " ( " dynasty)
  (let ((children nil)
        (mnames (starter mn))
        (fnames (starter fn)))
    (do ((n 0 (add1 n)))
        ((eq n tt))
      (setq father (getn mnames (get-rand p)))
      (setq mother (getn fnames (get-rand p)))
      (princ father)
      (princ mother)
      (cond ((and (not (eq father mother))
                  (not-closely-related father mother))
             (do ((i 0 (add1 i))
                  (stopp (get-rand 7)))
                 ((eq i stopp))

               (setq child (get-child  (get-rand p) mnames fnames))
               (when5 (homeless child father mother)
                      (setq children (cons child children))
                      (write-affirm (affirm father 'parent-of child) dynasty)
                      (write-affirm (affirm mother 'parent-of child) dynasty)
                      (make-male father)
                      (make-female mother)))
             (make-siblings children)
             (setq children nil))))))
  (princ " ) " dynasty)
  (close dynasty))



;; This function creates sibling relations given a list of names.
(defun make-siblings (children &aux (others children))
  (if (and children others)
      (do ((child (my-car children) (my-car children))
           ( children (my-cdr children) (my-cdr children)))
          ((null child))
          (do ((sib (my-car others) (my-car others))
               (others (my-cdr others) (my-cdr others)))
              ((null sib))
              (cond ((not (eq child sib))
                     (write-affirm (affirm child 'sibling-of sib) dynasty)))))))



;; This function prints the elements in an array (a).
(defun format-array (a)
  (do ((n 0 (add1 n)))
      ((eq n (array-length a)))
    (print (aref a n))))
```

```
;; TESTS

;; CHILDREN
;; Children: returns the list of children of a parent p.

;; Sequential Version.
(defun children (p &optional (pp *wildcard*))
  (pull-objects (matching-links  p 'parent-of pp)))

;; Combination #1
(defun p-children (p &optional (pp *wildcard*))
  (p-pull-objects (p-matching-links  p 'parent-of pp)))

;; Combination #2
(defun p2-children (p &optional (pp *wildcard*))
  (p-pull-objects (p2-matching-links  p 'parent-of pp)))

;; Combination #3
(defun p3-children (p &optional (pp *wildcard*))
  (p-pull-objects (p3-matching-links  p 'parent-of pp)))




;;SIBLINGS: returns the siblings of a person (c).

;; Sequential Version
(defun siblings (c &optional (pp *wildcard*))
  (pull-objects (matching-links c 'sibling-of pp)))

;; Combination #1
(defun p-siblings (c &optional (pp *wildcard*))
  (p-pull-objects (p-matching-links c 'sibling-of pp)))

;; Combination #2
(defun p2-siblings (c &optional (pp *wildcard*))
  (p-pull-objects (p2-matching-links c 'sibling-of pp)))

;; Combination #3
(defun p3-siblings (c &optional (pp *wildcard*))
  (p-pull-objects (p3-matching-links c 'sibling-of pp)))




;; COUSINS: returns the list of cousins of a person (c).

;; Sequential Version
(defun cousins (child)w
  (pull-objects
    (matching-links
      (pull-objects
        (matching-links
          (pull-subjects
            (matching-links *wildcard* 'parent-of child))
          'sibling-of *wildcard*))
      'parent-of *wildcard*)))


;; Combination #1
(defun p-cousins (child)
  (p-pull-objects
    (p-matching-links
      (p-pull-objects
        (p-matching-links
          (p-pull-subjects
            (p-matching-links *wildcard* 'parent-of child))
          'sibling-of *wildcard*))
      'parent-of *wildcard*)))
```

```
;; Combination #2
(defun p2-cousins (child)
  (p-pull-objects
    (p2-matching-links
      (p-pull-objects
        (p2-matching-links
          (p-pull-subjects
            (p2-matching-links *wildcard* 'parent-of child))
          'sibling-of *wildcard*))
      'parent-of *wildcard*)))



;; Combination #3
(defun p3-cousins (child)
  (p-pull-objects
    (p3-matching-links
      (p-pull-objects
        (p3-matching-links
          (p-pull-subjects
            (p3-matching-links *wildcard* 'parent-of child))
          'sibling-of *wildcard*))
      'parent-of *wildcard*)))




;; GRANDCHILDREN: returns the grandchildren of a person pp.


;; Sequential Version
(defun grandchildren (p &optional  (pp *wildcard*))
  (children (children p) pp))

;; Combination #1
(defun p-grandchildren (p &optional  (pp *wildcard*))
  (p-children (p-children p) pp))

;; Combination #2
(defun p2-grandchildren (p &optional  (pp *wildcard*))
  (p2-children (p2-children p) pp))

;; Combination #3
(defun p3-grandchildren (p &optional  (pp *wildcard*))
  (p3-children (p3-children p) pp))




;; Macros for ease of use.
(defmacro males () '(pull-subjects (matching-links *wildcard*  'sex 'male)))
(defmacro females () '(pull-subjects (matching-links *wildcard* 'sex 'female)))
(defmacro fathers () '(pull-subjects (matching-links (males) 'parent-of *wildcard*)))
(defmacro mothers () '(pull-subjects (matching-links (females) 'parent-of *wildcard*)))


(defmacro father (c) '(parents ,c (males)))
(defmacro mother (c) '(parents ,c (females)))
(defmacro sisters (c) '(siblings ,c (females)))
(defmacro brothers (c) '(siblings ,c (males)))
(defmacro sons (c) '(children ,c (males)))
(defmacro daughters (c) '(children ,c (females)))
(defmacro grandfathers (c) '(grandparents ,c (males)))
(defmacro grandmothers (c) '(grandparents ,c (females)))
(defmacro grandsons (c) '(grandchildren ,c (males)))
(defmacro granddaughters (c) '(grandchildren ,c (females)))


(defmacro count-kids (p) '(length (children ,p)))
(defmacro all-parents () '(pull-subjects (matching-links *wildcard* 'parent-of *wildcard*)))
```

```
;; This function returns the average number of children in a family.
;; Useful for evaluating the semantic network.
(defun avg-children (&optional (pp *wildcard*))
  (if (null pp) (setq pp (all-parents)))
  (let ((num 0))
    (cond ((listp pp)
           (do ((p (my-car pp) (my-car pp))
                (pp (my-cdr pp) (my-cdr pp)))
               ((null p))
             (setq num (+ num (count-kids p))))
           (/ num (length pp)))
          (t (count-kids pp)))))


;; THE BEST TESTS

(defun best-p-children (p &optional (pp *wildcard*))
  (pull-objects (p-matching-links  p 'parent-of pp)))

(defun best-p2-children (p &optional (pp *wildcard*))
  (pull-objects (p2-matching-links  p 'parent-of pp)))

(defun best-p3-children (p &optional (pp *wildcard*))
  (pull-objects (p3-matching-links  p 'parent-of pp)))


(defun best-p-siblings (c &optional (pp *wildcard*))
  (pull-objects (p-matching-links c 'sibling-of pp)))

(defun best-p2-siblings (c &optional (pp *wildcard*))
  (pull-objects (p2-matching-links c 'sibling-of pp)))

(defun best-p3-siblings (c &optional (pp *wildcard*))
  (pull-objects (p3-matching-links c 'sibling-of pp)))


(defun best-p-parents (c &optional (pp *wildcard*))
  (pull-subjects (p-matching-links pp 'parent-of c)))

(defun best-p2-parents (c &optional (pp *wildcard*))
  (pull-subjects (p2-matching-links pp 'parent-of c)))

(defun best-p3-parents (c &optional (pp *wildcard*))
  (pull-subjects (p3-matching-links pp 'parent-of c)))


(defun best-p-cousins (c &optional (pp *wildcard*))
  (p-children
    (p-siblings
      (pull-subjects
        (p-matching-links *wildcard* 'parent-of c))) pp))

(defun best-p2-cousins (c &optional (pp *wildcard*))
  (p2-children
    (p2-siblings
      (pull-subjects
        (p2-matching-links *wildcard* 'parent-of c))) pp))

(defun best-p3-cousins (c &optional (pp *wildcard*))
  (p3-children
    (p3-siblings
      (pull-subjects
        (p3-matching-links *wildcard* 'parent-of c))) pp))



(defun best-p-grandchildren (p &optional  (pp *wildcard*))
  (p-children (p-children p) pp))

(defun best-p2-grandchildren (p &optional  (pp *wildcard*))
  (p2-children (p2-children p) pp))

(defun best-p3-grandchildren (p &optional  (pp *wildcard*))
  (p3-children (p3-children p) pp))
```

```
;;; -*- Package: MULTILISP; Syntax: Common-lisp -*-
;;; This file contains the programs to run and test the school network.


;; The list of fall terms names.
(defvar *fall-term-names* '(fall80 fall81 fall82 fall83 fall84 fall85 fall86 ))

;; The list of sprint term names.
(defvar *spring-term-names* '(spr80 spr81 spr82 spr83 spr84 spr85 spr86 ))

;; The list of professor names.
(defvar *prof-names* '(jones smith halstead terman davis katz hulsizer king french
                       carrol lewis feld  corbato matttuck tucker
                       kingery weitzman saloner eckaus fisher dornbusch
                       kaledin yurek wuensch kalanji toomre eager ogilie modgliani
                       roylance wnek sadaway ring witt rosensweig moran gifford sussman
                       abelson troxel ward helgason quillan macdonald scotti
                       beck von-novak vezza ford
                       zdonik caruso newman dewald iuliano roth kaplan reiche))

;; The list of course names.
(defvar *course-names* '(english  calculus physics chemistry geology history
                                  biology economics computers accounting literature
                                  psychology philosophy algebra algorithms ))

;; The list of possible grades.
(defvar *grade-list* '(A B C D F))

;; The list of possible class rooms.
(defvar *room-list*  '(3-321 34-101 26-100 10-250 4-120 4-149 9-423 43-418 51-114
                       51-329 13-100 26-231 38-166 38-500 12-266 18-362 26-111))


;; Free variables.
(defvar *rooms* nil)
(defvar *profs* nil)
(defvar *fall-terms* nil)
(defvar *spring-terms* nil)
(defvar *grades* nil)
(defvar *num-fterms* (length *fall-term-names*))
(defvar *num-sterms* (length *spring-term-names*))
(defvar *num-profs*  (length *prof-names*))
(defvar *filename* "vx:/usr/hjb/school.links")

;; This function randomly assigns terms to a class (c) and write the
;; link onto the file (ff).
(defun assign-terms (c ff &aux (nn *num-fterms*))
  (let ((result nil)
        (link nil))
    (do ((i 0 (add1 i)))
        ((eq i (- nn 2)))
        (setq link (write-affirm (affirm c 'term (getn *fall-terms* (get-rand nn))) ff))
        (if (not (member link result))
            (setq result (cons link result))))
    (do ((i 0 (add1 i)))
        ((eq i (- nn 2)))
        (setq link (write-affirm (affirm c 'term (getn *spring-terms* (get-rand nn))) ff))
        (if (not (member link result))
            (setq result (cons link result))))
    (starter result)))

;;  This function randomly assigns professors  to classes given in c-array.
;;  and write the links onto the file (ff).
(defun assign-prof (c-array ff)
  (let ((result nil)
        (temp nil))
    (do ((i 0 (add1 i)))
        ((eq i (array-length c-array)))
        (setq temp (write-affirm (affirm (getn *profs* (get-rand *num-profs*))
                                         'teaches
                                         (getn c-array i)) ff))
        (if (not (member temp result))
            (setq result (cons temp result))))
    (starter result)))
```

```
;; This function randomly assigns courses to a number of students denoted
;; by numstu. Courses given in ccs. The links are written onto the file (ff).
(defun assign-stu-course (numstu ccs ff)
  (let ((fem (starter f-names))
        (mal (starter m-names))
        (result nil)
        (mlink  nil)
        (flink nil)
        (numc (array-length ccs)))
    (do ((i 0 (add1 i)))
        ((eq i numstu))
        (setq flink (write-affirm
                     (affirm (getn fem i) 'course (getn ccs (get-rand numc))) ff))
        (setq mlink (write-affirm
                     (affirm (getn mal i) 'course (getn ccs (get-rand numc))) ff))
        (if (not (member mlink result))
            (setq result (cons mlink result)))
        (if (not (member flink result))
            (setq result (cons flink result))))
    (starter result)))



;; This function randomly assigns rooms to designated classes given in
;; prof-course. The links are written onto the file (ff).
(defun assign-rooms (prof-course ff)
  (let ((nums (array-length prof-course))
        (num-rooms (array-length *rooms*)))
    (do ((i 0 (add1 i)))
        ((eq i nums))
        (write-affirm
          (affirm (getn prof-course i) 'room (getn *rooms* (get-rand num-rooms))) ff))))



;; This function randomly assigns grades to designated students and classes
;; given by stu-course. The links are written onto the file (ff).
(defun assign-grades (stu-course ff)
  (let ((nums (array-length stu-course)))
    (do ((i 0 (add1 i)))
        ((eq i nums))
        (write-affirm (affirm (getn stu-course i) 'grade (getn *grades* (get-rand 5))) ff))))



;; This function randomly generates data for a school data base.
;; With relations such as: term course grade (class-)room professor.
(defun set-up (&optional (numx 61) &aux classes profs studs school)
  (setq school (outfile *filename*))
  (princ "( " school)
  (setq *profs* (starter *prof-names*))
  (setq *fall-terms* (starter *fall-term-names*))
  (setq *spring-terms* (starter *spring-term-names*))
  (setq *rooms* (starter *room-list*))
  (setq *grades* (starter *grade-list*))
  (do ((cc (my-car *course-names*) (my-car *course-names*))
       (*course-names* (my-cdr *course-names*) (my-cdr *course-names*)))
      ((null cc))
      (setq classes (assign-terms cc school))
      (setq profs (assign-prof classes school))
      (setq studs (assign-stu-course numx classes school))
      (assign-rooms profs school)
      (assign-grades studs school))
  (princ" ) " school)
  (close school))
```

```
;; VERSION TESTS

;;  Test I
(defun class-prof (class term)
  (pull-subjects (matching-links *wildcard*
                                 'teaches
                                 (matching-links class 'term term))))

(defun p-class-prof (class term)
  (p-pull-subjects (p-matching-links *wildcard*
                                 'teaches
                                 (p-matching-links class 'term term))))

(defun p2-class-prof (class term)
  (p-pull-subjects (p2-matching-links *wildcard*
                                 'teaches
                                 (p2-matching-links class 'term term))))

(defun p3-class-prof (class term)
  (p-pull-subjects (p3-matching-links *wildcard*
                                 'teaches
                                 (p3-matching-links class 'term term))))

;; Best cases for test I


(defun best-p-class-prof (class term)
  (pull-subjects (p-matching-links *wildcard*
                                 'teaches
                                 (p-matching-links class 'term term))))

(defun best-p2-class-prof (class term)
  (pull-subjects (p2-matching-links *wildcard*
                                 'teaches
                                 (p2-matching-links class 'term term))))

(defun best-p3-class-prof (class term)
  (pull-subjects (p3-matching-links *wildcard*
                                 'teaches


                                 (p3-matching-links class 'term term))))
;;  Test II

(defun stu-prof (stu)
  (pull-subjects (matching-links *wildcard*
                                 'teaches
                                 (pull-objects (matching-links stu 'course *wildcard*)))))

(defun p-stu-prof (stu)
  (p-pull-subjects (p-matching-links *wildcard*
                                 'teaches
                                 (p-pull-objects (p-matching-links stu 'course *wildcard*)))))

(defun p2-stu-prof (stu)
  (p-pull-subjects (p2-matching-links *wildcard*
                                 'teaches
                                 (p-pull-objects
                                   (p2-matching-links stu 'course *wildcard*)))))

(defun p3-stu-prof (stu)
  (p-pull-subjects (p3-matching-links *wildcard*
                                 'teaches
                                 (p-pull-objects
                                   (p3-matching-links stu 'course *wildcard*)))))
```

```
;; Best cases for Test II

(defun best-p-stu-prof (stu)
  (pull-subjects (p-matching-links *wildcard*
                                   'teaches
                                   (pull-objects (p-matching-links stu 'course *wildcard*)))))

(defun best-p2-stu-prof (stu)
  (pull-subjects (p2-matching-links *wildcard*
                                    'teaches
                                    (pull-objects
                                      (p2-matching-links stu 'course *wildcard*)))))

(defun best-p3-stu-prof (stu)
  (pull-subjects (p3-matching-links *wildcard*
                                    'teaches
                                    (pull-objects
                                      (p3-matching-links stu 'course *wildcard*)))))



;; Test III

(defun st-grades (stu class term)
  (format-links (matching-links (matching-links stu
                                                'course
                                                (matching-links class 'term term))
                                'grade
                                *wildcard*)))

(defun p-st-grades (stu class term)
  (format-links (p-matching-links (p-matching-links stu
                                                    'course
                                                    (p-matching-links class 'term term))
                                  'grade
                                  *wildcard*)))

(defun p2-st-grades (stu class term)
  (format-links (p2-matching-links (p2-matching-links stu
                                                      'course
                                                      (p2-matching-links class 'term term))
                                   'grade
                                   *wildcard*)))

(defun p3-st-grades (stu class term)
  (format-links (p3-matching-links (p3-matching-links stu
                                                      'course
                                                      (p3-matching-links class 'term term))
                                   'grade
                                   *wildcard*)))
```

References

[1] Anderson, T.L., *The Design of a Multiprocessor Development System,* M.I.T. Laboratory for Computer Science TR-279, 1982.

[2] Brachman, R.J., "On the Epistemological Status of Semantic Networks", prepared in part at Bolt Beranek and Newman Inc. under contracts supported by the Defense Advance Research Projects Agency and the Office of Naval Research.

[3] Brachman, R.J., "What ISA and isn't: An analysis of taxonomic links in semantic networks," *IEEE Computer, Special Issue on Knowledge Representation*, 1983.

[4] Brachman, R.J., R.E. Fikes, and H.J. Levesque, "KRYPTON: A functionsal approach to knowledge representation," *IEEE Computer, Special Issue on Knowledge Representation*, 1983.

[5] Gray, S. L., "Using Futures to Exploit Parallelism in Lisp," M.S. thesis, M.I.T. Department of Electrical Engineering and Computer Science, 1986.

[6] Halstead, R.H., "Multilisp: A Language for Concurrent Symbolic Computation," *ACM Transactions on Programming Languages and Systems 7:4,* Oct. 1985, pp. 501-538.

[7] Katz, B., and P. H. Winston, "Parsing and Generating English Using Commutative Transformations," A.I. Memo No. 667, Artificial Intelligence Laboratory of M.I.T., 1982.

[8] Winston, P. H., *Artificial Intelligence, 2nd. Ed.*, Addison-Wesley, Reading, MA. 1984.