Report Number: WUCS-89-08

1989-06-01

User's Manual for CCRC: (Common Lisp Version) Computing Reference Classes Statistical Reasoning Shell v. 2.5

R. P. Loui

CCRC implements a subset of Kyburg's rules for statistical inference. The system states from 1961 and is briefly described in "The Reference Class," (H. Kyburg Philosophy of Science 50, 1982). Consult the paper "Computing Reference Classes" (R. Loui, in Kanal, L. and Lemmer, J., Uncertainty in AI, v.1, North-Holland 1987) for a precis of the ideas underlying this program. This document is only the skeleton of a manual. It is designed to get the novice on the program as quickly as possible, and to provide some guidance for advanced questions. This piece of software is the extended version of... **Read complete abstract on page 2.**

User's Manual for CCRC: (Common Lisp Version) Computing Reference Classes Statistical Reasoning Shell v. 2.5

R. P. Loui

Complete Abstract:

CCRC implements a subset of Kyburg's rules for statistical inference. The system states from 1961 and is briefly described in "The Reference Class," (H. Kyburg Philosophy of Science 50, 1982). Consult the paper "Computing Reference Classes" (R. Loui, in Kanal, L. and Lemmer, J., Uncertainty in AI, v.1, North-Holland 1987) for a precis of the ideas underlying this program. This document is only the skeleton of a manual. It is designed to get the novice on the program as quickly as possible, and to provide some guidance for advanced questions. This piece of software is the extended version of a prototype principally intended to assist AI research on reasoning with uncertainty. This program is a small prototype extended so that it can be patched into larger experimental systems.

USER'S MANUAL FOR CCRC: (COMMON LISP VERSION) COMPUTING REFERENCE CLASSES, STATISTICAL REASONING SHELL v. 2.5

R. P. Loui

WUCS-89-08

June 1989

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

ABSTRACT

CCRC implements a subset of Kyburg's rules for statistical inference. The system dates from 1961 and is briefly described in "The Reference Class," (H. Kyburg, Philosophy of Science 50, 1982). Consult the paper "Computing Reference Classes" (R. Loui, in Kanal, L. and Lemmer, J., Uncertainty in AI, v. 1, North-Holland 1987) for a precis of the ideas underlying this program.

This document is only the skeleton of a manual. It is designed to get the novice on the program as quickly as possible, and to provide some guidance for advanced questions.

This piece of software is the extended version of a prototype principally intended to assist AI research on reasoning with uncertainty.

This program is a small prototype extended so that it can be patched into larger experimental systems.

User's Manual for CCRC: (Common) Computing Reference Classes
Statistical Reasoning Shell
Software version 2.6, 11/21/88

R. Loui
Manual version 2.1

1. First Glances.

1.0. Getting Started.

To get started, enter kcl and load either "prob.l" (interpreted) or "prob" (compiled).

1.1. Parts Inventory.

```
size filename
   5 arith.lisp*
   2 globals.lisp*
   9 kbstuff.lisp*
   4 kyburg.lisp*
   2 printing.lisp*
   1 prob.l*
  18 probkerns.lisp*
   7 relstats.lisp*
   3 sets.lisp*
  19 struct.lisp*
   2 xp.lisp*
```

1.2. Sketch.

CCRC implements a subset of Kyburg's rules for statistical inference. The system dates from 1961 and is briefly described in "The Reference Class," (H. Kyburg, Philosophy of Science 50, 1982). Consult the paper "Computing Reference Classes" (R. Loui, in Kanal, L. and Lemmer, J., Uncertainty in AI, v. 1, North-Holland 1987) for a precis of the ideas underlying this program.

This document is only the skeleton of a manual. It is designed to get the novice on the program as quickly as possible, and to provide some guidance for advanced questions.

1.3. History.

This piece of software is the extended version of a prototype principally intended to assist AI research on reasoning with uncertainty. Version 1, originally called IND begun in 1984 in Franz, was based on backward-chaining and proved unwieldy even for small problems. Version 2.1, begun a year later, was completed overnight and did no deductive inference. To it was added a forward chainer and counting, 2.2, and it was used in the UNIXEX testbed for network usage prediction. 2.3 was the conversion to Allegro Common on the MacIntosh which internalized the counting and consolidated the interface. 2.4 was a revision in Kyoto Common that altered data structures and added functionality. 2.5 (planned) adds more restrictions on combinatorics.

Thus, this program is a small prototype extended so that it can be patched into larger experimental systems.

1.4. Use.

According to "Evidential Reasoning Compared in a Network Usage Prediction Testbed" (R. Loui, Uncertainty Workshop IV, 1988) Kyburg's system is competitive among uncertainty methods that must do evidence combination. Its major virtues are intuitive explanations of why a probability is the value calculated (because we can always point to a reference class), and its ability to combine a variety of sources, some of which might be sampling, some of which might be subjective.

Use this system for uncertain reasoning with few but varied statistical sources. It is not as good as other methods at crunching large data quickly. Query time should be on the order of a few seconds to a few minutes. It is a good method when we have to make the best of the few data that we have.

1.5. Example.

The system is designed to take a knowledge base such as

```
(setq KBASE '(
              (% ( home win ) (.4 .5))
              (MEM wg home)
              ))
```

which says that wg (wednesday's game) is a member of the class home, and the frequency of wins among home games is the interval [.4, .5], then take a query such as

```
(prob 'wg 'win)
```

and compute the value (.4 .5) for the probability.

1.6. Extended Examples.

The succession of examples that follows is fairly self-explanatory. I will annotate only briefly. In a later section, I discuss two more complex examples in more detail. The present section is designed just to give an idea of the inputs and outputs in the program's most basic mode.

The first example shows the most basic query, which is almost the

same as above. "no dis" says that there was no disagreement between the candidate and the challenger.

```
(new-kbase)
(setq KBASE '(
              (% ( home win ) (.3 .8))
              (MEM wg home)
              ))
(print KBASE)
(print (P :x 'wg :Z 'win))

<       new candidate % HOME (.30 .80)
no dis > % HOME (.30 .80)

((0.3 0.8))
```

The next example is shows reasoning from two non-conflicting statistical sources. (I ...) is the operator for intersecting sets. "add" has to do with combinations of statistical sources, which result in sets formed with the "XP" operator.

```
(new-kbase)
(setq KBASE '(
              (% ( home win ) (.3 .8))
              (% ( night win ) (.4 .6))
              (MEM wg (I home night))
              ))
(print KBASE)
(print (P :x 'wg :Z 'win))

        &(0 1)HOME NIGHT
 add; (.22 .86)

        &(0 1)HOME NIGHT
 add; (.22 .86)

<       new candidate % NIGHT (.40 .60)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .86)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .86)
no dis > % HOME (.30 .80)
no dis > % NIGHT (.40 .60)

((0.4 0.6000000000000001))
```

This example shows reasoning when there are two conflicting sources.

```
(new-kbase)
(setq KBASE '(
              (% ( home win ) (.3 .8))
              (% ( night win ) (.4 .9))
              (MEM wg (I home night))
              ))
(print KBASE)
(print (P :x 'wg :Z 'win))
```

```
        &(0 1)HOME NIGHT
 add; (.22 .97)

        &(0 1)HOME NIGHT
 add; (.22 .97)

<       new candidate % HOME (.30 .80)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % HOME (.30 .80)
failed > % NIGHT (.40 .90)

<       new candidate % NIGHT (.40 .90)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
failed > % HOME (.30 .80)

<       new candidate % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % (XP (I NIGHT) (I HOME)) (.22 .97)
no dis > % HOME (.30 .80)
no dis > % NIGHT (.40 .90)

((0.2222222222222222 0.9729729729729729))
```

This example introduces statistical sources that are summaries of sampling from classes. "interval" indicates that samples were converted to intervals at the (default) acceptance level. "pass" indicates that a combination of sources was contemplated, but considered redundant.

```
(new-kbase)
(setq KBASE '(
              (s% ( home win ) (10 7))
              (s% ( night win ) (6 4))
              (s% ( (I home night) win ) (2 1))
              (MEM wg (I home night))
              ))
(print KBASE)
(print (P :x 'wg :Z 'win))

 interval (10 7) at 0.9 is (.45 .87)
 interval (6 4) at 0.9 is (.36 .88)
 interval (2 1) at 0.9 is (.13 .87)

        &(0 1)HOME NIGHT
 add; (.32 .98)

        &(0 2)HOME (I HOME NIGHT)
 pass;

        &(1 2)NIGHT (I HOME NIGHT)
 pass;

        &(0 1 2)HOME NIGHT (I HOME NIGHT)
 pass;
```

```
<       new candidate % HOME (.45 .87)
no dis > % (XP (I NIGHT) (I HOME)) (.32 .98)
no dis > % HOME (.45 .87)
no dis > % NIGHT (.36 .88)
no dis > % (I HOME NIGHT) (.13 .87)

((0.4539573930541266 0.8675294821361947))
```

The next example introduces the idea of recorded data, which may be aggregated into sampling information. Each REC statement is a datum, with some known properties. "counting" says what properties are being matched with each REC statement, and thereafter follows line-by-line a report of which properties matched. "reflect" has to do with a disagreement that may be ignored because of a subset relation. The query is processed twice, the second time to show the lack of distinction between "home" and "(home)" in version 2.6 of this program.

```
(new-kbase)
(setq KBASE '(
              (REC (recnum r001) (home t) (win t))
              (REC (recnum r002) (win t))
              (REC (recnum r003) (home t) (night t) (win f))
              (REC (recnum r004) (night t) (win t))
              (REC (recnum r005) (home t) (night t) (win f))
              (REC (recnum r006)  (win t))
              (REC (recnum r007) (night t) (win f))
              (MEM wg (I home night))
              ))
(print KBASE)
(print (P :x 'wg :Z 'win :count))
(print (P :x 'wg :Z '(win) :count))

--->counting (((I HOME NIGHT) WIN))

((NIGHT NIL))

((NIL (WIN)))

(((I NIGHT HOME) NIL))

((NIGHT (WIN)))

(((I NIGHT HOME) NIL))

((NIL (WIN)))

((HOME (WIN)))
 interval (4 1) at 0.9 is (.06 .63)
 interval (2 0) at 0.9 is (0.0 .55)
 interval (7 4) at 0.9 is (.30 .81)
 interval (3 1) at 0.9 is (.08 .73)

        &(0 1)NIGHT (I NIGHT HOME)
 pass;

        &(0 2)NIGHT NIL
```

```
pass;

       &(1 2)(I NIGHT HOME) NIL
pass;

       &(0 3)NIGHT HOME
add; (.01 .82)

       &(1 3)(I NIGHT HOME) HOME
pass;

       &(2 3)NIL HOME
pass;

       &(0 1 2)NIGHT (I NIGHT HOME) NIL
pass;

       &(0 1 3)NIGHT (I NIGHT HOME) HOME
pass;

       &(0 2 3)NIGHT NIL HOME
pass;

       &(1 2 3)(I NIGHT HOME) NIL HOME
pass;

<      new candidate % NIGHT (.06 .63)
no dis > % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % NIGHT (.06 .63)
failed > % (I NIGHT HOME) (0.0 .55)

<      new candidate % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % NIGHT (.06 .63)
failed > % (I NIGHT HOME) (0.0 .55)

<      new candidate % (I NIGHT HOME) (0.0 .55)
reflect > % (XP (I HOME) (I NIGHT)) (.01 .82)
reflect > % NIGHT (.06 .63)
no dis > % (I NIGHT HOME) (0.0 .55)
reflect > % NIL (.30 .81)
reflect > % HOME (.08 .73)

((0.0 0.5497713746075072))


--->counting (((I HOME NIGHT) (WIN)))

((NIGHT NIL))

((NIL ((WIN))))

(((I NIGHT HOME) NIL))

((NIGHT ((WIN))))
```

```
(((I NIGHT HOME) NIL))

((NIL ((WIN))))

((HOME ((WIN))))
 interval (4 1) at 0.9 is (.06 .63)
 interval (2 0) at 0.9 is (0.0 .55)
 interval (7 4) at 0.9 is (.30 .81)
 interval (3 1) at 0.9 is (.08 .73)

        &(0 1)NIGHT (I NIGHT HOME)
 pass;

        &(0 2)NIGHT NIL
 pass;

        &(1 2)(I NIGHT HOME) NIL
 pass;

        &(0 3)NIGHT HOME
 add; (.01 .82)

        &(1 3)(I NIGHT HOME) HOME
 pass;

        &(2 3)NIL HOME
 pass;

        &(0 1 2)NIGHT (I NIGHT HOME) NIL
 pass;

        &(0 1 3)NIGHT (I NIGHT HOME) HOME
 pass;

        &(0 2 3)NIGHT NIL HOME
 pass;

        &(1 2 3)(I NIGHT HOME) NIL HOME
 pass;

<       new candidate % NIGHT (.06 .63)
no dis > % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % NIGHT (.06 .63)
failed > % (I NIGHT HOME) (0.0 .55)

<       new candidate % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % (XP (I HOME) (I NIGHT)) (.01 .82)
no dis > % NIGHT (.06 .63)
failed > % (I NIGHT HOME) (0.0 .55)

<       new candidate % (I NIGHT HOME) (0.0 .55)
reflect > % (XP (I HOME) (I NIGHT)) (.01 .82)
reflect > % NIGHT (.06 .63)
no dis > % (I NIGHT HOME) (0.0 .55)
reflect > % NIL (.30 .81)
reflect > % HOME (.08 .73)
```

```
((0.0 0.5497713746075072))
```

1.7. Data Structure Sketch.

The data structures can be very simple and are only converted into more efficient structures when the first query is made. So the simplest applications can be run with concern only for KBASE, a list of statements. Forward chaining inference can be added with IBASE, a list of forward chaining rules. There are three good ways to add to KBASE. They are (acc-k), (faster-acc-k), and (fastest-acc-k). There is one good way to add to IBASE. It is (acc-i). KBASE will eventually be copied and split into pieces, called MEMBASE, IFFBASE, ACCBASE, %BASE, and RECBASE. It is (kbase-sub) that does this dirty work, and it does so automatically whenever, say, an IFF statement is sought and IFFBASE has not yet been constructed. Since this is automatic, only KBASE need be stored and restored. However, it is insufficient to (setq KBASE nil); instead, one should use (clear-kbase-only), which sets all the lists to nil.

IBASE will eventually be copied and split into two pieces, called FINDBASE and UNIFYBASE. FINDBASE stores all the FIND rules in a recursive hash table. UNIFYBASE is just a list that contains AND rules and UNIFY rules and should be kept short for efficiency. (relevant-rules) is the best way to access something in FINDBASE.

1.8. Kyburg Sketch.

Kyburg's procedure starts with frequency data in the form of intervals, for example,

```
(% ( (weekend) (logged-on jackson) ) (.2 .4) ).
```

Sampling data must be converted into frequencies. This is done by approximating narrowest binomial confidence intervals at some level of confidence. Some of these intervals can be combined on purely set-theoretic grounds. For example,

```
(% ( (in-use castor) (logged-on jackson) ) (.3 .5) ).
```

can be combined with the interval above to give

```
(% ((XP (weekend) (in-use castor)) (logged-on jackson))
        ( g(.2, .3) g(.4, .5) ) )
```

where (XP a b) is

> {<x, y> : a holds of x; b holds of y; and (logged-on jackson) holds of x iff it holds of y }
>
> (in the % statement, the target property, (logged-on jackson) should really be a cross product (X (logged-on jackson) (logged-on jackson)), but we just ignore this).

and

$$g(x, y) = xy/(1 - x - y + 2xy).$$

These constructions are discussed in the references given above (particularly Loui, 1987).

Of these structures that contain sets and associated intervals, one must give the probability. The one that does is called the reference structure, which contains the reference class and the probability interval. According to Kyburg, there is a subset of all these structures that is most interesting. A structure belongs to this subset if it "reflects" all others with which it "disagrees." A structure reflects another if its class is more specific. Two structures disagree when their intervals do not nest and are not equal. Among the strucutres that belong to this subset of all the structures, the reference structure is the one with the narrowest interval (one can prove that all the intervals of all the structures in this subset do nest).

If there are c properties given, there may be 2^c = n relevant frequency statements, and there may be as many s = 2^n XP constructions, though the number of XP constructions worth considering decreases quickly on average, as the number of relevant frequency statements grows, for fixed c. Kyburg's requirement is computed in O(s^2), though there should be a data structure allowing O(s log s), which keeps track of both the interval-narrowness order and the class-specificity order.

1.9. Complex Examples.

The following knowledge base is interpreted to mean that wg (wednesday's game) is played at home and at night, in the rain, and on natural turf. Of 13 night games whose outcome was observed, 8 have been wins. Of 3 home night games, 2 have been wins. Of 2 home night games in the rain, 1 has been a win. The general frequency of wins on natural turf is the interval [.4, .8].

Note that the terms wg, home, night, etc., belong to the user's language. Sometimes it is convenient to use (home) or (pitcher gooden) to refere to classes; sometimes it is better to use home or pitecher-gooden. These are choices to be made by the application designer.

```
(new-kbase)
(setq KBASE '(
              (s% (night win) (13 8))
              (s% ((I home night) win) (3 2))
              (s% ((I home night rain) win) (2 1))
              (% (n-turf win) (.4 .8))
              (MEM wg (I home night))
              (MEM wg (I rain n-turf))
             ))
(P :x 'wg :Z 'win :no-trace)
```

The query directs us to compute the probability of the sentence

```
(MEM wg win)
```

given this knowledge. The value returned is

```
((0.4028042236307067 0.7914687696131544)) .
```

Here is a more involved example that demonstrates much of the functionality. The understanding of this knowledge base and this computation is one of the goals of this document.

The (mapc 'acc-i ...) part specifies some rules for IBASE, the forward chaining production-like deductive reasoner. The first rule, for instance, says that whenever (home) is found, (n-turf) is added. The (defun ...) statement at the end defines the truth conditions for a predicate that the user wants introduced into the language; it specifies when win-p holds of a recorded game, when it fails to hold, and when there is insufficient information recorded about the game to determine its truth value.

In the KBASE are two new kinds of statements. The (nc%>s ...) statement says that this is statistical information that should not be coered into a hypothetical sample when counting recorded games to update statistical information. And if there is conflict with the results of counting recorded games,and creating a sample from this class, then this statistical statement should dominate. The second kind of hitherto unfamiliar statement is the (ACC-AT ...) statement. These simply say that the contained statement should be accepted at acceptance levels equal to or less than the specified value.

```
(new-kbase)
(mapc 'acc-i '(
      ((find (home))
       (add (n-turf)))
      ((and (find (home)) (find (night))) (add (home-night)))
      ))
(mapc 'acc-k '(
      (s% ( (home) (win) ) (3 2))
      (% ( (I (home) (night)) (win)) (.4 .6))
      (nc%>s (nil (win)) (.1 1))
      (MEM tg (I (home) (pitcher gooden)))
      (MEM wg (home))
      (REC (recnum r001) (pitcher gooden) (home t) (win t))
      (REC (recnum r002) (win t))
      (REC (recnum r003) (home t) (night t) (win f))
      (ACC-AT .8 (MEM tg (night)))
      (ACC-AT .7 (IFF (MEM wg (win)) (MEM wg (win-p gooden))))
      ))
(defun win-p (p r)
 (cond
  ((and (equal '(win t) (assoc 'win (cdr r)))
        (equal p (assoc 'pitcher (cdr r)))))
  ((and (assoc 'win (cdr r)) (assoc 'pitcher (cdr r)))
   'f)
  (t nil)
 ))
```

Consider the queries

```
(P :def 'tg '(win) :count :bind)
```

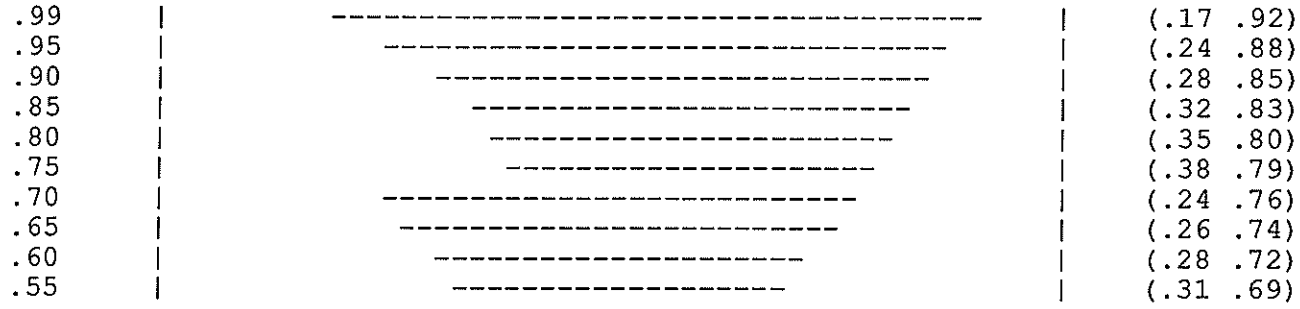
which asks for the probability that tg, tuesday's game, is in the set denoted by (win), asks that the probability be based on statistical information informed by counting any available records, and asks that the probability value be bound as a result; and

```
(P :def 'wg '(win) :graph :iter :count)
```

which asks for the probability that wg, wednesday's game, is in the set denoted by (win), asks that the records be counted, asks that the calculation be done iteratively over a list of acceptance levels (and uses the default list, since a list is not supplied here), and graphs each resulting probability interval.

They produce the following output, where the default value of TRACE was t. The output for the first query shows the result of counting REC statements in KBASE to update samples. For each record, it indicates the relevant properties that hold in the record that are common to the target individual. Also indicated is whether the target property held, reflected here when (win) or NIL appears. Next the output shows the conversion of samples into intervals at the default acceptance level, which had been set to .9. Then it shows the attempt to add for our consideration four combinations of statistical sources, all of which it chose not to add. It next shows how each candidate statistical source fared against other statistical sources, and finally shows the bound value which is the probability interval. The output for the second query graphs the probability intervals for various values of acceptance level.

```
...user command is...(P DEF 'TG '(WIN) COUNT BIND)

--->counting (((I (N-TURF) (HOME) (PITCHER GOODEN)) (WIN)))

(((I (N-TURF) (HOME)) NIL))

((NIL ((WIN))))

(((I (N-TURF) (PITCHER GOODEN) (HOME)) ((WIN))))
 interval (1 1) at 0.9 is (.29 1.0)
 interval (5 3) at 0.9 is (.28 .85)

        &(0 1)(I (N-TURF) (PITCHER GOODEN) (HOME)) NIL
 pass;

        &(0 2)(I (N-TURF) (PITCHER GOODEN) (HOME)) (I (N-TURF) (HOME))
 pass;

        &(1 2)NIL (I (N-TURF) (HOME))
 pass;

        &(0 1 2)(I (N-TURF) (PITCHER GOODEN) (HOME)) NIL (I (N-TURF)
                                                          (HOME))
 pass;

<       new candidate % (I (N-TURF) (PITCHER GOODEN) (HOME)) (.29 1.0)
no dis > % (I (N-TURF) (PITCHER GOODEN) (HOME)) (.29 1.0)
no dis > % NIL (.10 1)
reflect > % (I (N-TURF) (HOME)) (.28 .85)

((0.2905129316293615 1.0))

>
...user command is...(P DEF 'WG '(WIN) GRAPH ITER COUNT)
        0    .1   .2   .3   .4   .5   .6   .7   .8   .9   1.0
```


.99 | (.17 .92)
.95 | (.24 .88)
.90 | (.28 .85)
.85 | (.32 .83)
.80 | (.35 .80)
.75 | (.38 .79)
.70 | (.24 .76)
.65 | (.26 .74)
.60 | (.28 .72)
.55 | (.31 .69)
T

2. Syntax of Language.

2.0. Sets.

Sets are the basic building blocks. Note that the set of home games can be denoted by

```
home
```

or by

```
(home) .
```

In software version 2.6, if both forms are used, they denote the same set (this is a change from 2.5). The latter form is more useful when predicates are used or when there is an indicator function to determine membership. For example,

```
(pitcher gooden)
```

denotes the set of games in which gooden was the pitcher.

Intersections are denoted by I-prefixes

```
(I a b)
(I home night)
(I (home) (night) (pitcher gooden))
```

and certain subsets of cross products are used temporarily by the system, indicated by XP-prefixes,

```
(XP a b)
```

which denotes the set

```
{<x, y> : (AND (MEM x a) (MEM x b)
               (IFF (MEM x Z) (MEM y Z)))}
```

where Z is the current target property of a query.

(NOT a) is the recommended form of complement, but no inference is currently performed involving this prefix.

2.1. Statements in IBASE.

The idea has been to provide enough expressiveness for necessary forward chaining, but not so much that the system spends more time doing deductive inference than statistical inference. A more general unifier and forward chainer could be plugged in (see the functions associated with struct.lisp:(defun expand (pred-list) and struct.lisp:(defun match-bind (pred-list rule-car a-pred)).

Note that statements should be accepted into IBASE (using acc-i) before statements are accepted into KBASE (using acc-k) so that the proper forward chaining can be done at acc-k time.

Note that properties added by forward chaining are themselves queued for forward-chaining. Let's hope the chaining always terminates.

T rules.

ex. (T (ADD (always-true)))

says that (always-true) is added to all lists of properties.

syntax. (T (ADD [prop1]) [... other ADD exprs ...])

FIND rules.

ex. ((FIND (home)) (ADD (grass)) (ADD (deep-outfield)))

says that whenever (home) appears, add the properties (grass) and (deep-outfield).

So (home) becomes
(I (home) (grass) (deep-outfield)) and
(I (home) (night)) becomes
(I (home) (night) (grass) (deep-outfield)).

syntax. ((FIND [prop1]) (ADD [prop2]) [... other ADD exprs ...])
ref. struct.lisp: if-find-rule (rule)
effect of acc-i. also puts the rule in FINDBASE hashtable.

VAR-UNIFY rules.

ex. ((VAR-UNIFY (pitcher ?x)) (VAR-BIND (with ?x)))

says to add (with ?x) whenever (pitcher ?x) is found; where ?x is substituted appropriately.

So (I (pitcher gooden) (home)) becomes
(I (pitcher gooden) (with gooden) (home)).

ex. ((VAR-UNIFY (hit-batsman ?x ?y))
(ADD (imperfect-game)) (VAR-BIND (base-on-balls ?y)))

says to add (hit-batsman) and the result of binding (base-on-balls ?y) with the unifier of ?y, whenever (hit-batsman ?x ?y) is successfully unified.

So (hit-batsman gooden duncan) becomes
(I (base-on-balls duncan) (imperfect-game)
(hit-batsman gooden duncan)).

syntax. ((VAR-UNIFY (prop [...vars...])) [...ADD or VAR-BIND exprs...])

note. Variables must occur at the top level in a list. So (double-play-combo ?x backman) is ok, and so is (?x ?y) and (?x), but ?x is unacceptable, since it is not in a list, and so is (bob (went ?x)), since the variable occurs below the

top-level.

note. Variables in var-bind not used in var-unify are set to nil.

note. Currently, the first successful unification terminates attempted unification with subsequent properties on a list. So if

(I (hit-batsman gooden duncan) (hit-batsman myers gibson))

is expanded, the result is

```
(I (base-on-balls duncan) (imperfect-game)
        (hit-batsman gooden duncan)
        (hit-batsman myers gibson)).
```

This can be altered, but the shortcut saves inference time.

ref. struct.lisp: if-unify-rule (rule)

AND rules.

```
ex.  ( (AND (VAR-UNIFY (ss ?x)) (VAR-UNIFY (2b ?y)))
       (VAR-BIND (double-play-combo ?x ?y)) )
```

will convert (I (ss elster) (2b backman)) into
(I (double-play-combo elster backman) (ss elster) (2b backman)).

syntax. ((AND [... VAR-UNIFY or FIND exprs ...])
[...ADD or VAR-BIND exprs...])

note. unification fails if a variable is not bound to the same value in both expressions. So the rule
(AND (VAR-UNIFY (ss ?x)) (VAR-UNIFY (2b ?x)))
would fail on (I (ss elster) (2b backman)), and since there is no backtracking, it would also fail on
(I (ss elster) (2b backman) (2b elster)).

ref. struct.lisp: if-and-rule (rule)

Obviously, most of this forward-chaining can be improved by putting in a better forward-chainer, or by rewriting the system in PROLOG. The function (expand ...) and its sub-functions are all that need to be altered.

2.2. Statements in KBASE.

Statements in the knowledge base are currently of the following recognized types:

MEM statements.

---

ex. (MEM wg (night))

says that wg is an element of the set (night).

syntax. (MEM [x] [Y])
says that x is an element of Y.
ref. struct.lisp: if-mem-stmt (stmt)
acc-k. also puts on MEMBASE.

REC statements.

---

ex. (REC (recnum 001) (home t) (night t) (win f))

says that 001 is a game at night at home that was not a win.
This is our principal way of including sampling data.

syntax. A REC statment is the cons of 'REC and an association list.
ref. struct.lisp: if-record-stmt (stmt)
acc-k. also puts on RECBASE.

note. struct.lisp: rec-match (rnum)
will return the most recent REC statement
matching recnum rnum.

note. at the moment, REC statements are not individuated by
recnum, and recnum's need not appear, though this may
change in future versions.

note. the (cdr) of a REC statement is a record, also an assoc list,
and we will say that properties can hold of a record.

note. properties can hold of a record, t, their negations
can hold of a record, f, or there may be insufficient
data to tell if a property or its negation holds, nil.

note. to check if a property holds of a REC statement, we first
check if the property names a bound function. Examples are
(home) and (pitcher gooden) and (double-play-combo elster backman).

(home):
If there is no function bound to 'home, then it returns
the (assoc) of 'home on the record.

(pitcher gooden):
(pitcher gooden) is treated differently because it is a list.
If no function is bound to 'pitcher, then we do an (assoc) of
'pitcher on the record. If the slists match, return t; if
not, return f. If there is no association for 'pitcher in
the association list, return nil.

(double-play-combo elster backman):
If 'double-play-combo is bound, then we eval this s-expression
after appending the record as the third argument. The
definition of double-play-combo might look like this:

(defun double-play-combo (x y r)

```
                        (cond
                         ((and (equal x (cadr (assoc 'ss r)))
                               (equal y (cadr (assoc '2b r)))) 't)
                         ((and (assoc 'ss r) (assoc '2b r)) 'f)
                         (t nil)
                        )) .
```

(Yes, the 't is redundant, but improves readability). In fact, if 'pitcher had been bound, it could have been bound to

```
        (defun pitcher (x r)
                (cond
                 ((equal x (cadr (assoc 'pitcher r))) 't)
                 ((assoc 'pitcher r) 'f)
                 (t nil)
                ))
```

and had the same effect.

ACC-AT statements.

---

ex. (ACC-AT .7 (MEM wg (night)))

says that the MEM statement should be accepted at or below acceptance level .7.

syntax. (ACC-AT [lev] [statement])
ref. struct.lisp: if-acc-at-stmt (stmt)
acc-k. also puts on ACCBASE.

IFF statements.

---

ex. (IFF (MEM wg (win)) (MEM tg (win)))

says that wg is in (win) just in case tg is in (win).

syntax. (IFF [statement] [statement])
ref. struct.lisp: if-iff-stmt (stmt)
acc-k. also puts on IFFBASE.

note. The only useful IFF statements are those that connect two MEM statements. They are treated symmetrically, so (IFF s1 s2) need not accompany (IFF s2 s1).

STAT statements.

---

ex. (% (home win) (.3 .7))

says that the per cent of wins among the members of home is the interval [.3, .7].

note. when counting records to create samples, a % statement is

coerced to a hypothetical sample that would yield the reported bounds at the current acceptance level.

syntax. (% ([Y] [Z]) ([p] [q]))

ex. (s% ((home) (win)) (4 2))

says that a sample from (home) of size 4 yielded 2 elements of (win) and two elements of (NOT (win))

syntax. (s% ([Y] [Z]) ([s] [r]))

ex. (ks% ((I home night) win) (.8 (10 3)) (.7 (11 4)))

says that at or below .7, the sample 4 of 11 is appropriate for wins among home night games, and above that, but at or below .8, the sample of 3 of 10 is appropriate. These rules are defeasible, so when they both hold, the more specific rule (the former) takes precedence.

syntax. (ks% ([Y] [Z]) ([lev1] ([s1] [r1])) ([lev2] ([s2] [r2])) . . .)

note. lev's must be decreasing.

ex. (nc%>s ((I (home) (night)) (win)) (.3 .5))
ex. (nc%<s ((I (home) (night)) (win)) (.3 .5))
ex. (nc%=s ((I (home) (night)) (win)) (.3 .5))
ex. (nc%?s ((I (home) (night)) (win)) (.3 .5))

these statements say that the interval [.3, .5] reports the frequency of (win)s among (home) (night) games, and that these should not be coerced to hypothetical samples when adding relevant REC statements to samples (s%, ks%, and coerced % statements).

The operator (>, <, =, and ?) says what to do in relation to samples that are constructed by counting REC statements:

>s says to throw out any sample with the same sampling class and target property.
<s says to replace by any sample.
=s says to keep both and allow them to reflect each other.
?s says keep both and do not allow them to reflect each other.

syntax. (nc%[rel-op]s ([Y] [Z]) ([p] [q]))
ref. struct.lisp: if-stat-stmt (stmt)
acc-k. also puts on %BASE.

AND statements.

---

ex. (AND (MEM wg (home)) (MEM tg (I (home) (night))))
syntax. (AND s1 s2)

ref. struct.lisp: if-and-stmt (stmt)
acc-k. simply mapc's AND statements.

RESTRICT statements. (unimplemented)

ex. (RESTRICT (win) (((home) (night) (rain))) ((grass))))

which says that candidate reference classes for (win) are either intersections of combinations of (home), (night), and (rain), (2^3) or combinations of (grass), (2^1).

syntax. (RESTRICT [target-prop] ([list-1] [list-2] . . .))

3. Functions and Variables of Interest.

3.1. Interface Functions.

```
-----------------------------------------
functions for restarting
-----------------------------------------
kbstuff.lisp:  new-kbase-only ()

        + clears KBASE and all its sublists.

kbstuff.lisp:  new-ibase-only ()

        + clears IBASE and all its sublists.

kbstuff.lisp:  new-kbase ()
 or
kbstuff.lisp:  new-bases ()

        + clears KBASE and IBASE and sublists.

        + clears KBASE and IBASE and sublists.

kbstuff.lisp:  save-bases ()

        + writes KBASE on kbase.l and IBASE on ibase.l.

-----------------------------------------
functions for inference
-----------------------------------------
kbstuff.lisp:  acc-i (rule)

        + accepts a forward chaining rule into IBASE.

        + rule syntax appears in 2.1.

kbstuff.lisp:  forward-chain (s)

        + forward chains a statement using current IBASE.

-----------------------------------------
functions for knowledge
-----------------------------------------
kbstuff.lisp:  acc-k (stmt)

        + accepts a statement into KBASE and its sublists.

        + does forward chaining, uses include to check for redundancy,
                and checks to see if KBASE should be split.

kbstuff.lisp:  faster-acc-k (stmt)

        + accepts a statement into KBASE and its sublists.

        + does forward chaining, uses cons with no check for redundancy,
```

and checks to see if KBASE should be split.

kbstuff.lisp: fastest-acc-k (stmt)

+ accepts a statement into KBASE and its sublists.

+ does no forward chaining, uses cons with no check for redundancy, and does not check to see if KBASE should be split.

---

functions for querying

---

probkerns.lisp: P (&rest args)

+ calculates the probability of a statement with several options, which may occur in any order, and all of which have default values. The options are

```
:query, :prob, :def, :default, :x, :Z,
:given, :gensym,
:bind, :print, :show, :graph,
:lev, :level, :max-combs,
:count,
:iter, :lev-list, :std-iter,
:trace, :major-sections, :why,
:auto, :augment, :no-auto, :no-augment
```

and are explained next.

:query, :prob, :def, :default, :x, :Z, :given, :gensym are ways of specifying what query is to be calculated.

- follow :query and :prob with a MEM statement, '(MEM x Z).
  ex. (P :query '(MEM wg win))
- follow :def and :default with an x and a Z.
  ex. (P :def 'wg 'win)
- follow :x with an x.
  ex. (P :x 'wg)
  (it will use the previous target property if none is specified)
- follow :Z with a Z.
  ex. (P :x 'wg :Z 'win)
  ex. (P :Z 'win)
  (it will use the previous target individual if none is specified)
- follow :given with a list of properties being given, or else an intersection of properties, or else a MEM statement, or an AND statement. If not a MEM statement, it is assumed this is a Y, and we accept the statement `(MEM ,x ,Y). :given affects KBASE only temporarily.

```
                        ex.
                        (P :def 'win 'wg :given '(I home night))
                        ex.
                        (P :def 'win 'wg :given '(home night))
                        ex.
                        (P :def 'win 'wg :given '(MEM wg home))
                - :gensym says the same as :x (gensym).
                        It is a useful way to generate an
                        arbitrary individual quickly,
                        in conjunction with :given.
                        ex.
                        (P :Z 'win :gensym :given '(home night))
                - if query information is missing,
                        the previous query's x and/or Z
                        are substituted.  So (P) makes
                        sense if x and Z are understood;
                        it's not the exact query as
                        before; rather, it is the default
                        query with the last x and Z.

        :bind, :print, :show, :graph
                are ways of specifying what to do with the
                        result.  The default is to bind.
                        :graph is assumed on :std-iter.
                        :print and :show are the same thing.

        :lev, :level, :max-combs
                set level and max-combs temporarily.
                - follow :lev or :level with a real in
                        [0, 1].  Actually, levels (.5, 1]
                        are the ones that make sense.
                - follow :max-combs with an integer;
                        effective range is 3 to 6 or maybe 7.

        :count
                says to include counts of REC
                        statements for this query.
                - note that the default is NOT to count.

        :iter, :lev-list, :std-iter
                says to iterate the calculation.
                - follow :lev-list with a list of
                        levels, such as '(.99 .98 .5).
                - if a :lev-list is not provided,
                        the standard iter list is used.

        :trace, :major-sections, :why
                sets trace temporarily.
                - describes major computations as they
                        proceed.
                - producs a (why) output after the query
                        is processed.

        :auto, :augment, :no-auto, :no-augment
                temporarily forces the flag for augmenting
                        KBASE with ACC-AT results of queries.
```

```
        + some examples:
                (P :bind :iter :count :trace :given '(g1 g2)
                   :prob 't1 :gensym)
                (P :bind :no-trace :print :given '(MEM t1 Y1) :lev .99
                   :prob '(MEM t1 Z1))
                (P :show :level .66 :x 't1 :Z 'Z1)
                (P :def 't1 'Z1)
                (P ':given '(home) :gensym)
                (P x '(win))
                (P :count)
                (P :std-iter :count)
                (P :lev-list '(.99 .8 .54) :graph)

    probkerns.lisp:  why ()

        + attempts to do a trace of why the reference class is
                in fact the reference class; takes BEST-STAT
                and checks for reflection and disagreement with
                RELEVANT-STATS.

        + will work only after a query.

    probkerns.lisp:  why-not (class)

        + attempts to explain the failure of the inference
                structure based on the class supplied.

        + will work only after a query.

    ----------------------------------------
    additional functions
    ----------------------------------------
    kbstuff.lisp:  close-kbase ()

        + finds all the meaningful queries in KBASE and
                attempts to find their probabilities, iterating
                over levels.

        + a very ambitious computation which you should expect to
                have to interrupt.

    kbstuff.lisp:  forward-chain-kbase ()

        + mapcar's 'forward-chain on KBASE and
                sets the result to KBASE.

        + this resets the sub-bases, e.g. ACCBASE.
```

3.2. Environment Functions.

```
    arith.lisp:  best-interval (s r lev)

        + returns the narrowest interval at lev for a sample of r
```

from s.

arith.lisp: invert-interval (pair lev)

+ attempts to invert the interval at the level, e.g., (.4 .7) at .9 inverts to (24 13), since 13 out of 24 at .9 yields the interval (.39 .69); (.4 .7) at .8 inverts to (11 6) because 6 out of 11 at .8 yields (.38 .70).

kbstuff.lisp: lev-additions ()

+ returns detached ACC-AT sentences that are acceptable at the current ACC-LEVEL.

kbstuff.lisp: find-queries ()

+ finds meaningful queries in KBASE.

kyburg.lisp: if-undefeated (stat slist)

+ t if stat is undefeated in slist.

kyburg.lisp: if-defeated (stat slist)

+ t if stat is defeated in slist.

kyburg.lisp: if-stat-reflects (stat1 stat2)

+ t if (set-of stat1) reflects (set-of stat2).

kyburg.lisp: best-of (stat-list)

+ calculates the best stat among stat-list; that is, essentially, finds the strongest undefeated stat.

probkerns.lisp: prob (x Z)

+ old form of (P :def x Z)

probkerns.lisp: count-prob (x Z)

+ old form of (P :def x Z :count)

probkerns.lisp: GP (given-set prop)

+ old form of (P :gensym :Z prop :given given-set)

probkerns.lisp: GN (given-set prop)

+ old form of (P :gensym :Z prop :given given-set :count)

probkerns.lisp: match-class (class stat-list)

+ finds the stat in stat-list with (set-of) matching class.

probkerns.lisp: match-YZ (YZ stat-list)

```
        + finds the stat in stat-list with ((set-of) (prop-of))
                matching YZ.

relstats.lisp:  find-relevant-stats (x Z)

        + finds relevant stats for x and Z.

relstats.lisp:  XP-augment (source-list)

        + adds interesting combinations of source-list stats.

sets.lisp:  if-subset (set1 set2)

        + checks to see if set1 is subset of set2.

sets.lisp:  if-reflects (set1 set2)

        + checks if set1 reflects set2, including XP forms of sets.

struct.lisp:  contains many useful functions for breaking apart
        sentences, and changing forms, etc., such as
                make-set (x)
                x-of (stmt)
                Y-of (stmt)
                YZ-of (stat)
                set-of (stat1)
                prop-of (stat1)
                make-stat (stmt)
                create-stat (YZ-info)
                reverse-flatten (sexp)
                no-order (a b) t)

struct.lisp:  rec-match (rnum)

        + returns the first REC statement with (recnum rnum).

struct.lisp:  incr-stat (stmt incr)

        + increments s% stmt with incr, where incr is usually
                '(1 1) or '(1 0).

struct.lisp:  bound-pair-of (stat1)

        + computes interval of all kinds of stat statements
                at current ACC-LEVEL, including s% and ks% stmts.

struct.lisp:  props-holding-here (count-pair this-rec)

        + determines which props in count-pair,
                e.g. ( ((home) (night)) (win) )
                hold in this-rec.

struct.lisp:  determine-tvalue (prop rec)

        + determines whether prop is t, f, or nil (indeterminate)
```

on a rec stmt.

struct.lisp: expand (pred-list)

+ forward chains a list of predicates.

struct.lisp: relevant-rules (trigger)

+ for trigger, such as '(FIND (pitcher gooden)),
returns possibly relevant in FINDBASE hash table.

xp.lisp: xp-two-bounds (bound1 bound2)

+ mapcar's the xp function, g, to the bounds.

xp.lisp: xp-stats (stat1 stat2)

+ produces the XP combination of two stats.

3.3. Global Variables of Interest.

This is how globals.lisp: initializes all of the global variables.

```
(setq TRACE t)
     ; TRACE controls trace output

(setq ACC-LEVEL .9)
     ; acceptance level (not affected by probk)
(setq MAX-COMBS 3)
     ; maximum number of XP-combs
(setq ADD-COMBS t)
     ; add XP-combs
(setq AUTO-AUGMENT t)
     ; augment KBASE when statement is acceptable at a level
(setq AUTO-AUGMENT-WRITE nil)
     ; write augments of KBASE to kbase.l

(setq ITER-CALC nil)
     ; flag that is set when iter-kernel is being used -- see verbose()
(setq CLOSE-KBASE-TRACE t)
     ; trace on close-kbase fn

(setq BEST-STAT '(% (nil nil) (0 1)))
     ; most recent best-stat1
(setq RELEVANT-STATS nil)
     ; most recent relevant-stats
(setq XP-STATS nil)
     ; most recent xp-stat list
(setq RELEVANT-STATS-BEFORE nil)
     ; most recent stats after counting, but before posting of nc%>s's
(setq KB-STAT-LIST nil)
     ; most recent stats from KBASE prior to counting
(setq A-KBASE nil)
     ; most recent result of (augment-KBASE)
(setq MSPEC-CLASS nil)
```

```
        ; most recent find-most-spec-class calculation

(setq INCLUDE-STATS-ON-COUNTS t)
     ; includes % and s% (and n% ...) stmts on count-prob queries
(setq FORWARD-CHAIN-ALL nil)
     ; expands form of sets all the time, not just on query input or
            acc-k

(setq KBASE nil)
        (setq %BASE nil)
        (setq MEMBASE nil)
        (setq IFFBASE nil)
        (setq ACCBASE nil)
        (setq RECBASE nil)
(setq IBASE nil)
        (setq FINDBASE (make-hash-table))
        (setq UNIFYBASE$$$started$$$ nil)
        (setq UNIFYBASE nil)

(setq X nil)
(setq Z nil)
        ; last x and Z in queries

(defun verbose ()
        (and TRACE (not ITER-CALC))
        )

(setq MAJOR-SECTION-TRACE nil)
        ; prints beginning of major sections of computation
```

3.4. All Functions.

The following is a list of ALL the functions currently defined, of which there are 238.

3.4.1. By Package.

```
arith:  fact (x)
arith:  short-comb (a b)
arith:  binomial-summand (s r p)
arith:  normal-z-for (alpha)
arith:  interpolate (x table)
arith:  interpolate-sub (x last remain)
arith:  linear-interp (x x1 y1 x2 y2)
arith:  one-less-half (alpha)
arith:  k-alpha (lev)
arith:  x-bar (s r)
arith:  +-minus-interval (base vary)
arith:  best-interval (s r lev)
arith:  sq (x)
arith:  invert-interval (pair lev)
arith:  dirty-invert-inv (lb ub lev)
arith:  solve-n (m d k)
arith:  solve-r (n m)
arith:  invert-inv (lb ub lev)
arith:  err-fn (pair1 pair2)
```

```
arith:  bisect-on-d (m d lev lo hi)
arith:  width (x)
arith:  mid (x y)
arith:  bisect (hi-fn lo hi)
globals:  verbose ()
kbstuff:  lev-additions ()
kbstuff:  lev-filter-KBASE (stmt)
kbstuff:  acc-if-level-high-enough (stmt-acc-level real-stmt)
kbstuff:  possibly-augment (x Z ran-stat)
kbstuff:  try-to-augment (acc-level x Z lbound)
kbstuff:  poorer-lev (stmt x Z min-level)
kbstuff:  better-lev (stmt x Z min-level)
kbstuff:  find-queries ()
kbstuff:  find-queries-sub (kbase-remains queries)
kbstuff:  more-queries (mem-pair queries)
kbstuff:  find-target-props (Z kbase-remains props)
kbstuff:  possibly-add (queries x Y-list)
kbstuff:  close-kbase ()
kbstuff:  close-kbase-k (query-list lev-list)
kbstuff:  do-all-queries (query-list)
kbstuff:  save-bases ()
kbstuff:  acc-k (stmt)
kbstuff:  faster-acc-k (stmt)
kbstuff:  fastest-acc-k (stmt)
kbstuff:  forward-chain (s)
kbstuff:  forward-chain-kbase ()
kbstuff:  try-to-insert (x Z min-level)
kbstuff:  kbase-sub (name-of-sub-base filter-fn)
kbstuff:  new-kbase-only ()
kbstuff:  new-ibase-only ()
kbstuff:  new-kbase ()
kbstuff:  new-bases ()
kbstuff:  ibase-sub (name-of-sub-base)
kbstuff:  acc-i (rule)
kbstuff:  hash-find-rule (rule fbase)
kbstuff:  include-unify-rule (rule)
kbstuff:  recursive-hash (rule remain-key current-table)
kyburg:  if-undefeated (stat slist)
kyburg:  if-defeated (stat slist)
kyburg:  if-stat-reflects (stat1 stat2)
kyburg:  if-dis (stat1 stat2)
kyburg:  if-stronger (stat1 stat2)
kyburg:  best-of (stat-list)
kyburg:  best-of-r (ordstr checklist)
kyburg:  best-of-retrospective (stat1 remainlist)
printing:  princt (x)
printing:  vprint (x)
printing:  print-new-candidate (stat1)
printing:  print-sets (select-list stat-source)
printing:  pr-print (stat)
printing:  pretty-print-pair (pair)
printing:  rbound-pair-of (stat)
printing:  show-legend ()
printing:  show-graph (r bounds)
printing:  round2 (x)
printing:  p-very-round2 (x)
```

```
printing:  p-very-short (x)
probkerns:  P (&rest args)
probkerns:  P-sub (result pending-levs kernel x z key-pri key-gra
        key-why lev-flag)
probkerns:  prob (x Z)
probkerns:  prob1 (pair)
probkerns:  probk (x Z k)
probkerns:  prob-kernel (x Z)
probkerns:  prob-kernel1 (pair)
probkerns:  prob-iter (x Z)
probkerns:  prob-iter-show (x Z)
probkerns:  iter (k-probfn x Z)
probkerns:  iter-sub (k-probfn x Z lev-list)
probkerns:  iter-show (kernel-probfn x Z)
probkerns:  iter-show-sub (kernel-probfn x Z lev-list)
probkerns:  prob-iter-sub (x Z lev-list)
probkerns:  prob-iter-show-sub (x Z lev-list)
probkerns:  prob-sub (x Z ran-stat)
probkerns:  generic-probk (x Z k)
probkerns:  generic-prob-kernel (x Z)
probkerns:  generic-common-kernel (x Z stats)
probkerns:  generic-show (x Z ran-stat)
probkerns:  why ()
probkerns:  why-not (class)
probkerns:  why-kernel (class)
probkerns:  why-not-kernel (class)
probkerns:  match-class (class stat-list)
probkerns:  match-YZ (YZ stat-list)
probkerns:  count-prob (x Z)
probkerns:  count-prob1 (pair)
probkerns:  count-probk (x Z k)
probkerns:  count-prob-iter (x Z)
probkerns:  count-prob-iter-show (x Z)
probkerns:  count-prob-kernel (x Z)
probkerns:  count-prob-list-check (x Z lev-add count-pairs
        count-pairs-remain)
probkerns:  count-prob-ex (x Z lev-add count-pairs)
probkerns:  data-prop-extract (count-pairs rec-list)
probkerns:  prob-aggregate (x Z data-list-count count-pairs lev-add)
probkerns:  before-and-after-stat (one-list)
probkerns:  post-after-stats (after slist)
probkerns:  count-into-kb-stat-list (count-pairs data-list-count
        agg-list-virgin)
probkerns:  augment-agg-list (count-pairs data-list-count agg-list)
probkerns:  maybe-incr (one-stat datum-for-this-pair)
probkerns:  if-sub-con-pre (x y)
probkerns:  GP (given-set prop)
probkerns:  GN (given-set prop)
probkerns:  G (given-set prob-fn prop)
probkerns:  G-sub (given-set prob-fn x prop)
probkerns:  given (condition prob-query)
probkerns:  given-sub (condition prob-query)
probkerns:  new-given-sub (condition)
relstats:  find-bicond-pairs (x Z lev-add so-far)
relstats:  find-relevant-stats-pre (x Z)
relstats:  find-relevant-stats (x Z lev-add)
```

```
relstats:  find-relevant-stats-sub (x Z lev-add)
relstats:  find-relevant-stats-mspec (count-pairs lev-add)
relstats:  relevant-stat-list-check (Z most-spec-class lev-add)
relstats:  relevant-stat-list (Z lev-add most-spec-class so-far)
relstats:  find-most-spec-class (x lev-add so-far)
relstats:  XP-augment (source-list)
relstats:  also-add-maximal-XP (result source-list)
relstats:  maybe-add-this-comb (select-list source-list result)
relstats:  maximal-select-list (try-list source-list)
relstats:  maximal-sel-sub (whole-list untested source-list)
relstats:  maximal-selection (this remaining-tests source-list)
relstats:  add-stat-combs (choose stat-source add-to)
relstats:  add-this-comb (select-list stat-source add-to)
relstats:  this-comb (select-list stat-source result)
relstats:  add-or-pass (stat add-to)
sets:  if-subset (set1 set2)
sets:  if-onset (set1 set2)
sets:  share-prop (a b)
sets:  if-sub-con (list1 list2)
sets:  if-any-sub-con (p1 p2)
sets:  if-any-sub-all-con (list-list1 list-list2)
sets:  if-reflects (set1 set2)
sets:  make-explicit-xp (set1)
sets:  flat-intersect (set1 set2)
struct:  make-list-if-nec (x)
struct:  make-set (x)
struct:  if-and-stmt (stmt)
struct:  if-mem-stmt (stmt)
struct:  x-of (stmt)
struct:  Y-of (stmt)
struct:  if-rec-stmt (stmt)
struct:  if-record-stmt (stmt)
struct:  rec-match (rnum)
struct:  if-acc-at-stmt (stmt)
struct:  if-iff-stmt (stmt)
struct:  if-stat-stmt (stmt)
struct:  if-XP-stat-stmt (s)
struct:  YZ-of (stat)
struct:  set-of (stat1)
struct:  prop-of (stat1)
struct:  bound-pair-of (stat1)
struct:  right-pair-of (acc-at-list)
struct:  right-pair-of-sub (acc-remain result)
struct:  simple-bound-pair-of (stat1)
struct:  compute-bound-pair (pair level)
struct:  make-stat (stmt)
struct:  mem-stmt-match (x Y stmt)
struct:  categorical (stmt x Z)
struct:  non-intersection (set)
struct:  beginning-select (choose)
struct:  follow-select-r (choose inc)
struct:  increment-select (select-list from)
struct:  increment-carry (from remain build)
struct:  any-order (a b) t)
struct:  include (mem set)
struct:  list-member (mem set)
```

```
struct:  insert (mem set ordfn unique)
struct:  app-prop (key val propname)
struct:  mapsmart (fn-name v-list)
struct:  mapsmart-fast (fn-name v-list)
struct:  v-current (v-remain)
struct:  all-last-v (list-of-lists)
struct:  v-next (v-remain)
struct:  cdr-or-repeat (some-list)
struct:  props-holding (count-pairs this-rec)
struct:  props-holding-here (count-pair this-rec)
struct:  props-holding-one (maybe-list-pre this-rec)
struct:  determine-tvalue (prop rec)
struct:  special-extract (condition elist etrace)
struct:  extract (condition elist etrace)
struct:  create-stat (YZ-info)
struct:  null-stat (stmt)
struct:  incr-stat (stmt incr)
struct:  expand (pred-list)
struct:  expand-sub-first-time (pred-list result-list pending)
struct:  expand-sub-subsequent (pred-list result-list pending)
struct:  expand-sub-reenter (pred-list add-list result-list pending)
struct:  reverse-flatten (sexp)
struct:  reverse-flatten-r (remain result)
struct:  relevant-rules-first-time (trigger)
struct:  relevant-rules (trigger)
struct:  relevant-hash (remain-key current-table)
struct:  no-order (a b) t)
struct:  augment-list (old new exclusion-list)
struct:  expand-this (pred-list adds-so-far a-pred i-rules)
struct:  try-to-add (pred-list a-rule a-pred adds-so-far)
struct:  and-summarize (res-list)
struct:  combine-unifications (remain result)
struct:  combine-these (remain result)
struct:  if-and-rule (rule)
struct:  if-find-rule (rule)
struct:  if-unify-rule (rule)
struct:  if-and-rule-car (rule-car)
struct:  if-find-rule-car (rule-car)
struct:  if-unify-rule-car (rule-car)
struct:  match-bind (pred-list rule-car a-pred)
struct:  first-match-bind-sub (pattern pred-list)
struct:  match-bind-sub (pattern datum bindings)
struct:  is-new-variable (token bindings)
struct:  is-old-variable (token bindings)
struct:  is-variable (token)
struct:  try-to-add-sub (bindings rule-cdr adds-so-far)
struct:  add-till-empty (bindings rules adds-so-far)
struct:  replace-vars (bindings template)
struct:  replace-vars-sub (bindings remains so-far)
xp:  xp-two-sets (set1 set2)
xp:  insert-sub (head ins-list)
xp:  xp-two-bounds (bound1 bound2)
xp:  xp-two-bound-pairs (pair1 pair2)
xp:  xp-stats (stat1 stat2)
```

3.4.2. Sorted.

```
+-minus-interval (base vary) :arith
acc-i (rule) :kbstuff
acc-if-level-high-enough (stmt-acc-level real-stmt) :kbstuff
acc-k (stmt) :kbstuff
add-or-pass (stat add-to) :relstats
add-stat-combs (choose stat-source add-to) :relstats
add-this-comb (select-list stat-source add-to) :relstats
add-till-empty (bindings rules adds-so-far) :struct
all-last-v (list-of-lists) :struct
also-add-maximal-XP (result source-list) :relstats
and-summarize (res-list) :struct
any-order (a b) :struct
app-prop (key val propname) :struct
augment-agg-list (count-pairs data-list-count agg-list) :probkerns
augment-list (old new exclusion-list) :struct
before-and-after-stat (one-list) :probkerns
beginning-select (choose) :struct
best-interval (s r lev) :arith
best-of (stat-list) :kyburg
best-of-r (ordstr checklist) :kyburg
best-of-retrospective (stat1 remainlist) :kyburg
better-lev (stmt x Z min-level) :kbstuff
binomial-summand (s r p) :arith
bisect (hi-fn lo hi) :arith
bisect-on-d (m d lev lo hi) :arith
bound-pair-of (stat1) :struct
categorical (stmt x Z) :struct
cdr-or-repeat (some-list) :struct
close-kbase () :kbstuff
close-kbase-k (query-list lev-list) :kbstuff
combine-these (remain result) :struct
combine-unifications (remain result) :struct
compute-bound-pair (pair level) :struct
count-into-kb-stat-list (count-pairs data-list-count
        agg-list-virgin) :probkerns
count-prob (x Z) :probkerns
count-prob-ex (x Z lev-add count-pairs) :probkerns
count-prob-iter (x Z) :probkerns
count-prob-iter-show (x Z) :probkerns
count-prob-kernel (x Z) :probkerns
count-prob-list-check (x Z lev-add count-pairs
        count-pairs-remain) :probkerns
count-prob1 (pair) :probkerns
count-probk (x Z k) :probkerns
create-stat (YZ-info) :struct
data-prop-extract (count-pairs rec-list) :probkerns
determine-tvalue (prop rec) :struct
dirty-invert-inv (lb ub lev) :arith
do-all-queries (query-list) :kbstuff
err-fn (pair1 pair2) :arith
expand (pred-list) :struct
expand-sub-first-time (pred-list result-list pending) :struct
expand-sub-reenter (pred-list add-list result-list pending) :struct
expand-sub-subsequent (pred-list result-list pending) :struct
expand-this (pred-list adds-so-far a-pred i-rules) :struct
```

```
extract (condition elist etrace) :struct
fact (x) :arith
faster-acc-k (stmt) :kbstuff
fastest-acc-k (stmt) :kbstuff
find-bicond-pairs (x Z lev-add so-far) :relstats
find-most-spec-class (x lev-add so-far) :relstats
find-queries () :kbstuff
find-queries-sub (kbase-remains queries) :kbstuff
find-relevant-stats (x Z lev-add) :relstats
find-relevant-stats-mspec (count-pairs lev-add) :relstats
find-relevant-stats-pre (x Z) :relstats
find-relevant-stats-sub (x Z lev-add) :relstats
find-target-props (Z kbase-remains props) :kbstuff
first-match-bind-sub (pattern pred-list) :struct
flat-intersect (set1 set2) :sets
follow-select-r (choose inc) :struct
forward-chain (s) :kbstuff
forward-chain-kbase () :kbstuff
G (given-set prob-fn prop) :probkerns
G-sub (given-set prob-fn x prop) :probkerns
generic-common-kernel (x Z stats) :probkerns
generic-prob-kernel (x Z) :probkerns
generic-probk (x Z k) :probkerns
generic-show (x Z ran-stat) :probkerns
given (condition prob-query) :probkerns
given-sub (condition prob-query) :probkerns
GN (given-set prop) :probkerns
GP (given-set prop) :probkerns
hash-find-rule (rule fbase) :kbstuff
ibase-sub (name-of-sub-base) :kbstuff
if-acc-at-stmt (stmt) :struct
if-and-rule (rule) :struct
if-and-rule-car (rule-car) :struct
if-and-stmt (stmt) :struct
if-any-sub-all-con (list-list1 list-list2) :sets
if-any-sub-con (p1 p2) :sets
if-defeated (stat slist) :kyburg
if-dis (stat1 stat2) :kyburg
if-find-rule (rule) :struct
if-find-rule-car (rule-car) :struct
if-iff-stmt (stmt) :struct
if-mem-stmt (stmt) :struct
if-onset (set1 set2) :sets
if-rec-stmt (stmt) :struct
if-record-stmt (stmt) :struct
if-reflects (set1 set2) :sets
if-stat-reflects (stat1 stat2) :kyburg
if-stat-stmt (stmt) :struct
if-stronger (stat1 stat2) :kyburg
if-sub-con (list1 list2) :sets
if-sub-con-pre (x y) :probkerns
if-subset (set1 set2) :sets
if-undefeated (stat slist) :kyburg
if-unify-rule (rule) :struct
if-unify-rule-car (rule-car) :struct
if-XP-stat-stmt (s) :struct
```

```
include (mem set) :struct
include-unify-rule (rule) :kbstuff
incr-stat (stmt incr) :struct
increment-carry (from remain build) :struct
increment-select (select-list from) :struct
insert (mem set ordfn unique) :struct
insert-sub (head ins-list) :xp
interpolate (x table) :arith
interpolate-sub (x last remain) :arith
invert-interval (pair lev) :arith
invert-inv (lb ub lev) :arith
is-new-variable (token bindings) :struct
is-old-variable (token bindings) :struct
is-variable (token) :struct
iter (k-probfn x Z) :probkerns
iter-show (kernel-probfn x Z) :probkerns
iter-show-sub (kernel-probfn x Z lev-list) :probkerns
iter-sub (k-probfn x Z lev-list) :probkerns
k-alpha (lev) :arith
kbase-sub (name-of-sub-base filter-fn) :kbstuff
lev-additions () :kbstuff
lev-filter-KBASE (stmt) :kbstuff
linear-interp (x x1 y1 x2 y2) :arith
list-member (mem set) :struct
make-explicit-xp (set1) :sets
make-list-if-nec (x) :struct
make-set (x) :struct
make-stat (stmt) :struct
mapsmart (fn-name v-list) :struct
mapsmart-fast (fn-name v-list) :struct
match-bind (pred-list rule-car a-pred) :struct
match-bind-sub (pattern datum bindings) :struct
match-class (class stat-list) :probkerns
match-YZ (YZ stat-list) :probkerns
maximal-sel-sub (whole-list untested source-list) :relstats
maximal-select-list (try-list source-list) :relstats
maximal-selection (this remaining-tests source-list) :relstats
maybe-add-this-comb (select-list source-list result) :relstats
maybe-incr (one-stat datum-for-this-pair) :probkerns
mem-stmt-match (x Y stmt) :struct
mid (x y) :arith
more-queries (mem-pair queries) :kbstuff
new-bases () :kbstuff
new-given-sub (condition) :probkerns
new-ibase-only () :kbstuff
new-kbase () :kbstuff
new-kbase-only () :kbstuff
no-order (a b) :struct
non-intersection (set) :struct
normal-z-for (alpha) :arith
null-stat (stmt) :struct
one-less-half (alpha) :arith
P (&rest args) :probkerns
P-sub (result pending-levs kernel x z key-pri key-gra key-why
       lev-flag) :probkerns
p-very-round2 (x) :printing
```

```
p-very-short (x) :printing
poorer-lev (stmt x Z min-level) :kbstuff
possibly-add (queries x Y-list) :kbstuff
possibly-augment (x Z ran-stat) :kbstuff
post-after-stats (after slist) :probkerns
pr-print (stat) :printing
pretty-print-pair (pair) :printing
princt (x) :printing
print-new-candidate (stat1) :printing
print-sets (select-list stat-source) :printing
prob (x Z) :probkerns
prob-aggregate (x Z data-list-count count-pairs lev-add) :probkerns
prob-iter (x Z) :probkerns
prob-iter-show (x Z) :probkerns
prob-iter-show-sub (x Z lev-list) :probkerns
prob-iter-sub (x Z lev-list) :probkerns
prob-kernel (x Z) :probkerns
prob-kernel1 (pair) :probkerns
prob-sub (x Z ran-stat) :probkerns
prob1 (pair) :probkerns
probk (x Z k) :probkerns
prop-of (stat1) :struct
props-holding (count-pairs this-rec) :struct
props-holding-here (count-pair this-rec) :struct
props-holding-one (maybe-list-pre this-rec) :struct
rbound-pair-of (stat) :printing
rec-match (rnum) :struct
recursive-hash (rule remain-key current-table) :kbstuff
relevant-hash (remain-key current-table) :struct
relevant-rules (trigger) :struct
relevant-rules-first-time (trigger) :struct
relevant-stat-list (Z lev-add most-spec-class so-far) :relstats
relevant-stat-list-check (Z most-spec-class lev-add) :relstats
replace-vars (bindings template) :struct
replace-vars-sub (bindings remains so-far) :struct
reverse-flatten (sexp) :struct
reverse-flatten-r (remain result) :struct
right-pair-of (acc-at-list) :struct
right-pair-of-sub (acc-remain result) :struct
round2 (x) :printing
save-bases () :kbstuff
set-of (stat1) :struct
share-prop (a b) :sets
short-comb (a b) :arith
show-graph (r bounds) :printing
show-legend () :printing
simple-bound-pair-of (stat1) :struct
solve-n (m d k) :arith
solve-r (n m) :arith
special-extract (condition elist etrace) :struct
sq (x) :arith
this-comb (select-list stat-source result) :relstats
try-to-add (pred-list a-rule a-pred adds-so-far) :struct
try-to-add-sub (bindings rule-cdr adds-so-far) :struct
try-to-augment (acc-level x Z lbound) :kbstuff
try-to-insert (x Z min-level) :kbstuff
```

v-current (v-remain) :struct
v-next (v-remain) :struct
verbose () :globals
vprint (x) :printing
why () :probkerns
why-kernel (class) :probkerns
why-not (class) :probkerns
why-not-kernel (class) :probkerns
width (x) :arith
x-bar (s r) :arith
x-of (stmt) :struct
XP-augment (source-list) :relstats
xp-stats (stat1 stat2) :xp
xp-two-bound-pairs (pair1 pair2) :xp
xp-two-bounds (bound1 bound2) :xp
xp-two-sets (set1 set2) :xp
Y-of (stmt) :struct
YZ-of (stat) :struct

4. Explanation of Output.

Finally, let's try to understand the ouput on the queries above, line by line.
The query

```
(P :def 'tg '(win) :count :bind)
```

produces the output

```
--->counting (((I (N-TURF) (HOME) (PITCHER GOODEN)) (WIN)))
                                        ; says that this is the
                                        list of count-pairs being
                                        counted on RECBASE.

(((I (N-TURF) (HOME)) NIL))

((NIL ((WIN))))

(((I (N-TURF) (PITCHER GOODEN) (HOME)) ((WIN))))

                                        ; apparently there were three REC
                                        statements, and for each, we show
                                        which properties held.

 interval (1 1) at 0.9 is (.29 1.0)
 interval (5 3) at 0.9 is (.28 .85)

                                        ; s% statements, the result of
                                        coercing % statements and updating
                                        statistical info by counting, is
                                        finally converted to intervals.

                                        ; next, we consider each
                                        combination of % statements, to
                                        see if we want to add their XP
                                        combination.  "add" means it was
                                        added; "pass" means it wasn't.
                                        &(0 1) refers to the combination
                                        being attempted, and the corresponding
                                        classes, whose combination is being
                                        contemplated, are printed.

       &(0 1)(I (N-TURF) (PITCHER GOODEN) (HOME)) NIL
pass;

       &(0 2)(I (N-TURF) (PITCHER GOODEN) (HOME)) (I (N-TURF) (HOME))
pass;

       &(1 2)NIL (I (N-TURF) (HOME))
pass;

       &(0 1 2)(I (N-TURF) (PITCHER GOODEN) (HOME)) NIL (I (N-TURF)
                                                         (HOME))
```

```
 pass;
```

; we now consider each statistical statement, in (partial) order of strength. For each candidate statistic, we print how it fares against all the other statistics, until a disagreement is found, or until it is discovered that this is the best stat.

```
<       new candidate % (I (N-TURF) (PITCHER GOODEN) (HOME)) (.29 1.0)
no dis > % (I (N-TURF) (PITCHER GOODEN) (HOME)) (.29 1.0)
no dis > % NIL (.10 1)
reflect > % (I (N-TURF) (HOME)) (.28 .85)

((0.2905129316293615 1.0))
```

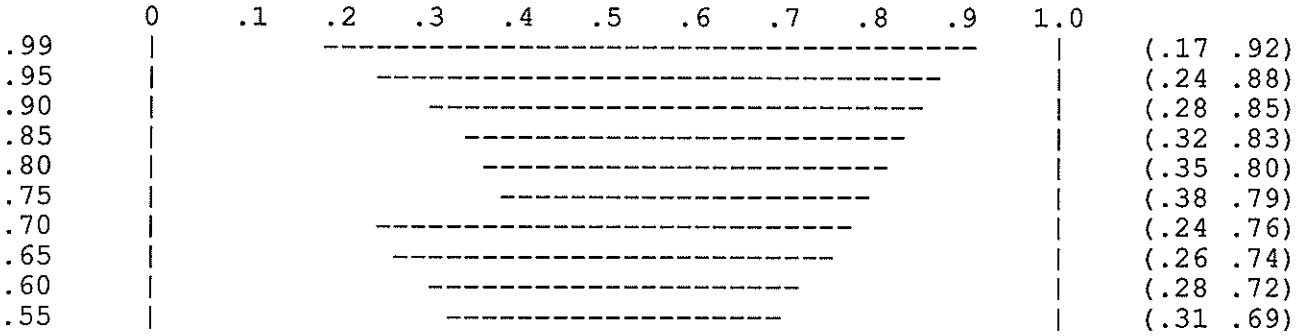
; the value is bound.

```
>
```

The query

```
(P :def 'wg '(win) :graph :iter :count)
```

produces the output




```
T
```

; this graphs the probability value at each ACC-LEVEL, and shows the numeric value. Note that if AUTO-AUGMENT is set, the probability is (1.0 1.0) when the lower bound on the probability at the last level exceeds the current ACC-LEVEL. This would be printed as a "+" instead of a "|" on the right border. This doesn't happen for (0.0 0.0) because we don't do inference with (NOT).

5. Further Information.

Questions and problems should be directed to loui@wucs1.wustl.edu, or kyburg@cs.rochester.edu.