A Guide to Cup

SIRAI, Hidetosi
School of Computer and Cognitive Sciences
Chukyo University
Tokodate 101, Kaizu-cho, Toyota, Aichi, 470--03 JAPAN
E-mail: sirai@sccs.chukyo-u.ac.jp

Version 0.80d (1993-09-01)

1. Introduction

CUP (or, cu-Prolog), which is an experimental constraint logic programming (CLP) language, originally developed for use on workstations such as SUN4. Unlike most conventional CLP systems, CUP allows user-defined predicates to be used as constraints, and is suitable for implementing a natural language processing system based on unification-based grammar formalisms. As an application of CUP, we have developed a parser based on JPSG theory (Japanese Phrase Structure Grammar) with the JPSG Working Group (chaired by Prof. Takao Gunji of Osaka University) at ICOT.

CUP was developed at ICOT. The base mechanism is called `Constraint Unification' (CU), and developed by Dr. Hasida. Sirai first implemented its prolog version, then Mr. Tsuda developed cu-Prolog entirely written in C. He and the author have revised it, and MacCup was written especially for Macintosh.

ICOT has the copyright of cu-Prolog, and distributes the source and binary object.

2. CUP Programming

2.1 Distinctive feature of CUP

The programming style is almost same with that of DEC10 Prolog. However CUP has several different features from it as follows:

(1) Constrained Horn Clause is supported.
(2) Partially Specified Term (PST) is supported.
(3) Operator `;' has a differnt meaning. It does NOT mean OR predicate. In CUP, it is used as a separator between the body part and the constraint part. Disjunctive control structure is written by using or/2, or/3, or/4, and or/5 predicates.

For example, the following program in usual Prolog

```
(a(X), a1(X), a2(X)) ; (b(X), b1(X)) ; c(X)
```

can be written in CUP as follows:

```
or( (a(X), a1(X), a2(X)), (b(X), b1(X)), (c(X)) )
```

or

```
or( [a(X), a1(X), a2(X)], [b(X), b1(X)], [c(X)] ).
```

2.2 How to Start and Quit

To start CUP, type the following:

```
            cup [CR]
      or
            cup <file_name> [CR]

    To quit it, there are three ways:
            CMND+Q
      or    %Q [CR]
      or    :-halt. [CR]
```

2.3 Memory Allocation of CUP

CUP uses the following data areas:

```
      SHEAP : (400K cells) storage area for user-defined programs
      HEAP  : (150K cells)  temporal storage area
      CHEAP : (800K cells) working area for processing constraints and
PSTs
      ENV   : (100K cells)  environment for variables' binding
      USTACK: (30K cells)   storage area for everything
      NAME  : (100K bytes)  storage area for strings
```

The numbers in the parenthes show their default sizes. However the user can change its size by providing extra arguments for `cup' command.

```
      cup [-s <n>] [-h <n>] [-c <n>] [-e <n>] [-u <n>] [-n <n>] [file]
```

2.4 Top level of CUP

At the top level of CUP, you will see a prompt `_'. Here you can do the following things:

(1) Write programs with/from the keyboard. All Horn clauses inputted from the keyboard are defined as user-defined predicates except that there are no system-defined predicates whose both name and arity are same.

e.g. `_ member(X,[X|_]).` ==> a definition of `member' is defined.

(2) Execute programs. When you input a sequence of literals following `:-', it is evaluated as a CUP program.

e.g. `_ :- member(X,[a,b,c]).` ==> evaluation of member.

(3) Transform constraints. When you input a sequence of literals following `@', it is transformed into some normal form.

e.g. `_ @ member(X,Y), append(Y,U,V).`

(4) Execute CUP commands. Some keywords following a `%' are dealt with as CUP commands.

2.5 Running CUP Programs

There are two standard ways when you develop CUP programs:

(1) Input CUP programs from the keyboard, run them, and debug them. After verifying them, save them into a file.

(2) Make a CUP program file with some editor such as Emacs. Then call CUP, load a program file, execute the program, debug it. After that, you can save it into a file, or edit the program file.

In order to load programs from a file, there are two ways:

(a) At the top level, input the name of file enclosed by double quote signs `"', or

(b) Providing the file's name for `cup' command.

To save a program into a file, there is the following way:
   (a) Use CUP command `%w' with file name.

   Sometimes you may lose a control over CUP. However there is a chance to recover the contorl by pushing `c' with the control key. Then you will see the following message:

```
Interrupt --- Input <T(race),C(ontinue),A(bort)>?
```

If you type C, CUP continues the current process.
If you type T, CUP goes into the step trace mode.
If you type A, CUP quits the current process, and returns to the top level.

2.6 Syntax of CUP Program

There are two types of Horn clauses in CUP, that is, conventional Horn clauses and Constrained Horn clauses. Furthermore, each Horn clause may be classified into three kinds as follows:

(1) Facts. They are written in the following way:

```
H.                or
H ; C1, ..., Cn.
```

e.g.
```
human(socrates).
lexicon(put, {pos/v, form/X, infl/Y}) ; agreement(X,Y).
```

(2) Rules. They are written in the following way:

```
H :- B1, ..., Bm.or
H :- B1, ..., Bm ; C1, ..., Cn.
```

e.g.
```
append([],X,X).
cat(Mother) :- cat(Left), cat(Right) ; psr(Mother,Left,Right).
```

(3) Queries. They are written in the following way:

```
:- B1, ..., Bm.        or
:- B1, ..., Bm ; C1, ..., Cn.
```

e.g.
```
:- permutation(X,[a,b,c]), well_orderd(X).
:- choice(X), good(X) ; memb(X, [a,b,c,d,e]).
```

3. CUP Commands

In this section, CUP commands are introduced. Most of them have the correspondents in CUP menu. In the below, PRED represents a predicate name, or a combination of the name and its arity such as psr/3, NUM represents a number, especially positive integer in most cases, and NAME represents a name atom.

3.1 General Commands

(1) %h Help. Shows the table of CUP commands.
(2) %d PRED Shows the definitions of the predicates PRED.
    %d * Shows the definitions of all predicates.
    %d ? Shows the names of predicates.
    %d - Shows the definitions of user-defined predicates.
(3) %f Shows the current status of memory usage.
(4) %Q Quits CUP.
(5) %R Initializes CUP.
(6) %c NUM Sets the maximum depth of refutations. If the process comes deeper than this, it fails.
(7) %u Switches the handling of undefined predicates between fail and error. Initially it is set to fail.

(8) %n NAME Sets the core of new generated names in the constraint transformation and/or the process of gensym. Initially it is set to `c'.

(9) %T Switches the timer between off and on. Initially it is ON. When it is on, each time a process is done, the consuming time (seconds) is shown.

(10) %I AREA/NUM (NOTE: only for UNIX version)
Increase the temporal work area by the designated number (note. the unit is K cell). And AREA should be either c, h, u, or e.
'c' means the work space for constraints and pst, 'h' means the heap, 'u' means the user stack, and 'e' means the environment save space. Thus when you write %I h/200,
the heap space will increased by 200K.

3.2 File I/O

(1) "File Name"
Loads a program from the file `File Name'.

(2) "File Name?
Loads a program from the file `File Name' with showing the content.

(3) %l Sets a log file.

(4) %w Writes the current defined programs into a file.

3.3 Debugging

(1) %p PRED Sets/Removes a spy flag onto the predeicate PRED.
%p * Sets spy flags onto all user defined predicates.
%p . Removes spy flags from all predicates.
%p > Sets/Removes a spy flag onto the constraint transformation process.
%p ? Shows the names of predicate which are set spy flags.

(2) %t Sets up/Cancels the trace mode.

(3) %s Sets up/Cancels the step trace mode.
The prompt tells what mode the system is. For example, the prompt is `_' in no trace mode, `$' in the trace mode, and `>' in the step trace mode.

3.4 Constraint Transformation

(1) %L Shows all generated predicates during the constraint transformation.

(2) %M NUM Sets the maximum number of variables used in one process of constraint transformation. In a pathological case, the constraint transformation may go into inifinite loop. By setting the maximum number of variables may prevent CUP into such a trouble.

(3) %P PRED Preprocesses PRED's constraint part, that is, the constraint
part of predicate PRED is transformed into `modular' form.
%P * Preprocesses constraint parts of all predicates.
%P ? Shows the definitions of predicates which have non-modular constraint parts.

(4) %o Sets the constraint transformation process into M-solvable mode. In this mode, the process checks the satisfiability

of the constraints. And if they are satisfiable, it transforms the constraints into semimodular form, where not every predicate may be modular. Otherwise, it fails.

(5) %a Sets the constraint transformation process into all-modular mode. In this mode, every generated predicate is made as modular.

(6) %S This is a toggle switch. The default is ON. If it is on, The constraints will be generated as simple as possible.

As the default, the constraint transformation process is set in M-solvable mode.

4. Built-in Predicates

4.1 Terms used in this guide

(1) term
atom, variable, complex term, partial specified term

(2) atom
constant, string, number

(3) constant
A sequence of characters which starts with lower case, or a sequence of any charcters which is enclosed by single quotes, or a sequence of special characters such as #, $, etc.

(4) string
A sequence of any characters including space which is enclosed by double quotes.

(5) number
integer, floating point number

(6) variable
A sequence of characters which starts with uppper case or underscore '_'. Variable whose name is '_' is called anonymous variable. Each anonymous variables is dealt with as different variables.

(7) complex term
p(t1, ..., tn) is called a compex term where 'p' is a constant, and t1, ..., tn are terms. Especially when it occurs as a term in some predicate, p(t1, ..., tn) is called as a function, otherwise it is called a predicate. And the 'p' is called a functor, the 'ti's arguments, and the 'n' is its arity.

(8) operator
Some functors can have prefix, postfix or infix notations. They are called operators. There are several pre-defined ones. And user can define more.

(9) PST (Partially Specified Term)
A sequence of pairs of a constant (called Feature Name) and a

term (called Feature Value) which is enclosed by braces { and } . Especially { } means a empty PST.
The pair of feature name and value is represented using the infix operator '/' such as Name / Value. And the pairs are separated with a comma ','.

(10) list
A sequence of terms enclosed by brackets [and]. Each term is separated by a comma ','. Especially [ ] means an empty list.
Formally a list is represented as a function whose functor is a period '.' and whose arity is 2. Thus [a,b,c] is a short-hand representation of .(a, .(b, .(c, []))).
Furthermore a function such as .(X,Y) is represented as [X | Y].

(11) file pointer
When reading characters or terms from a file, or writing characters or terms into a file, we need a structure called a file pointer.
In order to make a file pointer, use open/3 predicate. With the pointer as an argument to read/write predicate, we can open several files, read from them, and write into them.
It should be noted that 'user' is used as the input file from the keyboard and the output file to the screen.

(12) input/output stream
You can read characters or terms from a file, and write them into a file without file pointers. Only one you need is a stream which is set up by see (for input), tell and tella (for output).
Once a stream is set, the target of read/write become the stream file until the stream is closed by seen (for input) or told (for output). The default is the keyboard (fot input) and the screen (for output).

(13) clause
a sequence of terms enclosed by parentheses (and). Each term is separated by a comma ','.

4.2 List of built-in predicates

There are many built-in predicates in CUP. In the below, the arguments with + sign represent instantiated terms, and those with - sign represent free variables.

!
is same as `cut' in Prolog.

[File (, File1, ...)]
reads programs from the File(s). If there is - sign preceding the file name, the file should be read in the reconsulting way.

X+ < Y+
Arguments should be numbers. This succeeds if X < Y, otherwise

fails. This may be written as less(X,Y).
eg. :- 1.0 < 0.99. --> success
:- 1.0 < 1.00. --> success
:- 1.0 < 1.01. --> fail

X+ <= Y+

Arguments should be numbers. This succeeds if X <= Y, otherwise fails. This may be written as leq(X,Y).
eg. :- 1.0 <= 0.99. --> fail
:- 1.0 <= 1.00. --> success
:- 1.0 <= 1.01. --> success

X+ > Y+

Arguments should be numbers. This succeeds if X > Y, otherwise fails. This may be written as greater(X,Y).
eg. :- 1.0 > 0.99. --> success
:- 1.0 > 1.00. --> fail
:- 1.0 > 1.01. --> fail

X+ >= Y+

Arguments should be numbers. This succeeds if X >= Y, otherwise fails. This may be written as geq(X,Y).
eg. :- 1.0 >= 0.99. --> success
:- 1.0 >= 1.00. --> success
:- 1.0 >= 1.01. --> fail

X = Y

unifies X with Y. This may be written as equal(X,Y).

X == Y

succeeds if X is equal to Y, otherwise fails. By this predicate, you can check if two variables is truly equal. Same as eq(X,Y).
eg. :- X == X. --> success
:- X == Y. --> fail
:- a(b) == a(b). --> success
:- X=..[a,b], X == a(b). --> fail

Term =.. List

If the first argument, Term, is instantiated, this unifies the second with a list whose first member is Term's predicate name, and whose other members are Term's arguments.
Otherwise, that is, if Term is a free variable, List should be a list. This unifies Term with a term whose predicate name is List's first member and whose arguments are List's other members.

abolish(Name+,Arity+)

removes all the definitions of user-defined predicate, Name/Arity from CUP database.
eg. :- abolish(member,2). --> member/2 is removed.

arg(Pos+,Term+,Arg-)

The first argument, Pos, should be a list or a function with the arity more than or equal to one. Otherwise, this causes an error. This unifies the Pos-th argument of Term with Arg. When Term is a list such as [a,b,c], it is treated as a binary predicate like `.'(a,[b,c]).
eg. :- arg(2,a(b,c,d),X). --> X = c

assert(Head+)
assert(Head+, Body+)
assert(Head+, Body+, Constraint+)

registers a definition with Head, Body and Constraint in CUP database. Body and Constraint may be a clause or a list. When they are omitted, they are dealt with as nulls.
eg. :- assert(member(X,[_|Y]),(member(X,Y))).
--> member(X,[_|Y]) :- member(X,Y). is registered.

asserta(Head)
asserta(Head, Body)
asserta(Head, Body, Constraint)
is same as above.

assertz(Head)
assertz(Head, Body)
assertz(Head, Body, Constraint)
registers a definition with Head, Body and Constraint at the last in CUP database.

atomname(Term, String)
if Term is a functor, String is unified with Term's functor name (string). If String is a string, then Term is unified with an atom whose name (string) is equal to String.
eg. :-atomname(atom(X),Y). --> Y = "atom"
:-atomname(X,"string"). --> X = string

attach_constraint(X+)
attaches the constraint clauses, X, to the current process. X should be a clause or a list.
eg. :-attach_constraint((member(X,[a,b,c]))).
--> The value of X is constrained by member(X,[a,b,c]).

clause(Term+,Body,Constraint)
returns a definition which has a head unifiable with Term, a body unifiable with Body, and a constraint unifiable with Constraint. If such a definition has null body and/or null constraint, Body and/or Constraint are unified with a null list, []. This may unify every definition with a head unifiable with Term by causing backtracking.
eg. :- clause(member(X,Y),Body,Const).
--> If member(A,[A|_]) is defined, X, Y, Body and Const will be unified with A, [A|_], [], [], respcetively.

clause_list(Clause,List)
if Clause is not a variable, List is unified with a list whose elements are same as Clause's.
If List is a list, Clause is unified with a clause whose elements are same as List's. It should be noted that the list whose tail part is a variable like [A | X] is dealt as the list whose taile part is a list of a variable like [A,X].
eg. :-clause_list((a,b,c),X). --> X = [a,b,c]
:-clause_list(X,[a,b,c]). --> X = (a,b,c)

close(FP+)
closes the file designated by a file pointer, FP.

closefiles
closes all files opend by open/3.

compare(T1+,T2+,P)
Ti should be either a number or a string. This returns ==, < or > if T1 and T2 are equal, T1 is less than T2, or T1 is greater than T2, respectively.
eg. :- compare(123,124,X). --> X = `<'
:- compare("abc","ab",X). --> X = `>'

```
concat(A,B,C)
      unifies C with a string which will be got by concatenating two
      strings A and B. Any two of these arguments should be instantiated.
      You can get another solution by backtracking if it exists.
      eg. :- concat("ab",X,"abcd").           --> X = "cd"
          :- concat(X,Y,"abc").          --> X = "abc", Y = ""
            (causing backtracking)       --> X = "ab", Y="c"
concat2(String,List)
      If String is instantiated, this unifies List with a list whose
      members are characters composing the string.  Otherwise, if
      List is instantiated, this unifies String with a string which
      is composed by the members of List.
      eg. :- concat2("ab",X).                 --> X = ["a","b"]
          :- concat2(X,["hell", "o"]).  --> X = "hello"
condname(Terms+,List)
      The first argument, Terms, should be a list of terms.  This
      unifies List with a list whose members are predicate names appeared
      in Terms.
      eg. :- condname([c0(X),c1(Y,Z)], X).    --> X = [c0,c1])
consult(File)
        reads programs from the File.
count(N)
      If N is a number, the internal counter is set to it. If N is
      a variable, it is unified with the value of the counter.  After
      that, the counter is incremented by one.  Initially it is set to
      0.  When backtracking, this fails.
      eg. :- count(2), count(A), count(B).
      --> A = 2, B = 3.
default(Target+, Filter+, DefaultValue+)
      All arguments should be instantiated PSTs. This fails if Target
      doesn't have all attribute-value pairs in Filter. Otherwise,
      Target is unified with a PST whose attribute-value pairs
      are composed from those of DefaultValue which are compatible
      with Target.
      eg. :- X = {a / b, c / d}, default(X, {a / b}, {c / e, e / g}).
      --> X = {a / b, c / d, e / g}
          :- X = {a / b, c / d}, default(X, {a / c}, {e / g}).
      --> fail
divstr(String+,Number,FirstHalf,LastHalf)
      unifies the first Number character strings of String with
      FirstHalf, and the rest part of String with LastHalf.
      When Number may be negative integer, it is treated as the
      length of String `plus' Number.
      FirstHalf should be a character string.
      eg. :- divstr("abcde",-2,X,Y).          --> X = "abc", Y = "de"
          :- divstr("abcde",N,"abc",Y). --> N = 3, Y = "de"
          :-divstr("abcde",N,X,"de").         --> N = 3  X = "abc"
equal(X,Y)
      unifies X and Y. This is same as X = Y.
eq(X,Y)
      returns true if X is equal to Y. This is same as X == Y.
execute(List+)
      The argument should be either a list of terms or a clause. This
      evaluates each goal in the argument.  Failure of a goal causes
```

a backtracking.
eg. :- execute((memb(X,[a,b]), memb(X,[b,c]))).
--> X = b

fail
Always fails.

forall(Functor+,Clause+)
succeeds when for every solution for Functor satisfies Clause. Otherwise it fails.
eg. :- forall(append(X,Y,[a,b]),(write(X),tab,write(Y),nl)).
--> prints the following:
```
  []    [a,b]
  [a]   [b]
  [a,b] []
```
true (where X and Y are unbound)
:- forall(member(X,[a,b]),member(X,[c])).
--> fail

functor(Term,Name,Arity)
When the first argument, Term, is instantiated, it should be a list, a function, or a name atom. This unifies the predicate name of Term with Name, and its arity with Arity.
When Term is a free variable, both of Name and Arity should be instantiated. This unifies Term with a term whose predicate name is Name and whose arity is Arity.
eg.:- functor(a(b),X,Y). --> X = a, Y = 1
:- functor(X,a,3). --> X = a(_1,_2,_3)

gensym(Atom-)
unifies the argument, Atom, with a new generated name atom, which is composed some `core' name and an integer. The core name is initally set to `c', and you can change it by using CUP command `%n'.
eg. :- gensym(X). --> X = c0
%n new --> set the core name to `new'
:- gensym(X). --> X = new1

gensym(Name+, Atom-)
The second argument, Atom, should be either a name atom, or a string. This unifies the first argument, Name, with a new generated name atom.
eg. :- gensym(gen,X). --> X = gen2

geq(X+,Y+)
Arguments should be numbers. This succeeds if X >= Y, otherwise fails. This may be written as X >= Y.

get(X)
get(X,FP+)
gets a character code from the current input stream (the file pointer, FP). However a space, and control codes such as a carriage return is skipped. This unifies it with the first argument, X. When backtracking, this gets another character code from the input stream (file pointer).
X is unified with 255 if the end of file(EOF) is gotten.
eg. :- get(X), write(X).
--> X = 97, when you type `a' from the keyboard with a carriage return.

get0(X)
get0(X,FP+)

This is same as get/1,2 except that this gets a space and control codes such as a tab. Furthermore, X is unified with 31 for a carriage return code.

get1(X)

get1(X,FP+)

This is same as get0/1,2 except that carriage return code is not converted (ie. X is unified with 13).

greater(X+,Y+)

Arguments should be numbers. This succeeds if X > Y, otherwise fails. This may be written as X > Y.

halt

quits CUP.

isop(Prec,Type,Op-)

unifies a defined operator's precedence, type, and name with Prec, Type, Op, respectively. By backtracking, every operator can be shown.

```
eg. :- isop(X,Y,Z).          --> X = 900, Y = xfy, Z = '/'
    :- isop(X,Y,(:-)).       --> X = 1200, Y = fx
```

Note. The operator with high precedence (more than 1000) should be quoted as an argument.

length(X+,N)

The first argument, X, should be a list, a clause, a string, or a partially specified term. This unifies the second argument, N, with the number of elements of X (the number of characters in case that X is a string). If X is such a list as [A|_], then N is unified with 1.

```
eg. :- length([a,b,c],N).          --> N = 3
    :- length((a,b),N).                  --> N = 2
    :- length({a/1, b/2, c/X}, N).--> N = 3
```

leq(X+,Y+)

Arguments should be numbers. This succeeds if X <= Y, otherwise fails. This may be written as X <= Y.

less(X+,Y+)

Arguments should be numbers. This succeeds if X < Y, otherwise fails. This may be written as X < Y.

listing(Functor+)

listing(Functor+ / Arity+)

prints the definition(s) of the predicate whose name is equal to Functor's functor name. If Arity is not specified, all the definition(s) of the predicates are shown. It returns always true.

```
eg. :- listing(del).
--> prints the definitions of the predicate `del'.
    :- listing(a/3).
--> prints the definitions of the predicate `a' with the arity 3.
```

ml(Term,List)

is same as Term =.. List.

memb(X,Y)

The second argument, Y, should be a list. This unifies the first argument, X, with a memebr of Y. When backtracking, X will be unified with Y's other member. (This is the built-in member.)

```
eg. :- memb(X,[a,b,c]), write(X), fail.
--> `abcd' is typed out.
```

multiply(X,Y,Z)

This fails when more than or equal to two arguments are variables. This unifies the argument which is a free variable with some number so that X * Y = Z.

name(Atom,List)

If the first argument, Atom, is an name atom, the second argument, List, is unified with a list whose memebrs are character codes composing Atom.
If List is a list whose members are character codes (integer), Atom is unified with an name atom which is composed of the codes.

```
eg. :- name(gen, X).              --> X = [103, 101, 110]
    :- name(X, [97.98.99]).       --> X = abc
```

nl

nl(FP+)

types out a carriage return code, \r, onto the current output stream (the file pointer, FP).

not Term+

This fails if Term succeeds, otherwise succeeds.

```
eg. :- memb(X, [a,b,c]), not memb(X,[b,c,d]). --> X = a
    :- memb(X, [b,c]), not memb(X,[b,c,d]).   --> fail
```

op(Prec+,Type+,Op+)

If the third argument, Op, is a name atom, then it is defined as an operator with the precedence, Prec, and the type, Type. If Op is a list of name atoms, then each of them is defined as operators with the same precedence, Prec, and the same type, Type. The type should be one of among xf, yf, fx, fy, xfx, xfy and yfx. And the precedence should be integer, greater than 0 and less than 1200. You may re-define system defined operators such as :- except `,'.

```
eg. :- op(500, xfy, [to, from]).
```

--> it succeeds, and `to' and `from' are defined as operators whose precedence is 500 and whose type is xfy.

open(FileName, Mode+, FP)

opens a file whose name is designated by the first argument, Filename, for reading or writing, and unifies the file pointer with the third argument, FP. The second argument, Mode, should be either r, w, or a. The mode `r' means it open the file for reading. The mode `w' means it creates a new file (even if a file whose name is FileName) for writing. The mode `a' means it opens a possibly existing file for writing at end of file.

openfiles(FileName)

openfiles(FileName,FP)

openfiles(FileName,FP,Mode)

unifies the first argument, FileName, with a file name which is opened by open/3. Furthermore FP (and Mode) is/are unified with the file pointer (and its mode, respectively) if applicable. By backtracking, every opened file may be unified with FileName.

or(X+,Y+)

or(X+,Y+,Z+)

or(X+,Y+,Z+,U+)

or(X+,Y+,Z+,U+,V+)

Arguments should be a list of terms or a clause. This calls every goal in one argument sequentially until every goal succeeds. If it fails, another argument is evaluated. If every argument

returns failure, it fails.

pcon

writes the constraints attached to the current process.

pnames(PST+,List)

unifies the second argument, List, with the list whose members are the attribute name atoms of the partially specified term, PST.

```
eg. :- pnames({a / X, b / c}, Y).       --> Y = [a, b]
```

project_cstr(Term)

replaces the current constraints with a newly generated constraint which covers only the variables in the Term.

prompt(String+)

sets the prompt to the argument, String, which should be a string. This prompt is typed out when read/1 is called. Initially it is set to "? ".

```
eg. :- prompt("Enter> ").
--> The prompt is changed to "Enter> ".
```

put(X)

put(X,FP+)

types out the character code X onto the current output stream (the file pointer, FP).

```
eg. :- member(X,[97,98,99]),put(X),fail.
--> a, b, and c are typed out onto the screen.
```

pvalue(PST+,Pname+,Pvalue)

unifies the third argument, Pvalue, with the value whose attribute is Pname of the partially specified term, PST.

```
eg. :- X = {a / 1, b / 2}, pvalue(X, a, Y).   --> Y =    1
    :- X = {a / Y}, pvalue(X, a, c).          --> X = {a / c}
```

read(Term)

read(Term,FP+)

reads a term from the current input stream (the file pointer, FP), and unifies it with the first argument, Term. The term to be read should be delimited by a period. If CUP encounters the enf of file, Term will be unified with an special atom, end_of_file. If it reads from the keyboard, it puts a prompt, which is determined by prompt/1, onto the screen.

reconsult(File)

reads programs from the File. If there are any defined predicates which is also read from the File, the old one is abolished.

reset_timer

resets the timer. (cf. timer)

retract(Head+)

retract(Head+, Body)

retract(Head+, Body, Constraint)

removes a definition from CUP database whose head is unifiable with the first argument, Head, and whose body is unifiable with the second argument, Body, and whose constraint is unifiable with the third argument, Constraint. When backtracking, it may remove another definition. Body and Constraint are assumed null when they are omitted.

```
eg. :- regract(member(X,Y)).
```

--> if member/2 has such a definition as member(A,[A|_]), it will be removed, and X and Y are unified with A and [A|_], respectively.

see(File+)
      The argument, File, should be a string or a name atom. It sets
      the current input streamt to the file whose path name is File.
      If there is no such a file, it fails.
      eg. :- see("programs/test.p"), read(X), seen.
      --> CUP sets the current input stream to the `test.p' file of
      the `program' directory in the current directory, reads a term from
      it, and then closes the file.
seen
      closes the file which is the current input stream, and sets the
      current input stream to the keyboard.
stayin(Cond_Clause+,Pred_Clause+)
      If the result of Cond_Clause is true, then Pred_Clause will be
      evaluated in the unfolding process. Otherwise, it is neglected
      as constraints, and stay in as constraints until the Cond_Clause
        will become true in the future unfolding process.
strcmp(X+,Y+,P)
      The first and second arguments, X and Y, should be strings.
      This unifies the third argument, P, with either ==, < or > when
      X and Y are equal, X is less than Y, and X is greater than Y,
      respectively. (Note. The strings is lexicographically ordered.)
      eg. :- strcmp("ab", "abc", X).   --> X = '<'
          :- strcmp("abc", "ab", X).   --> X = '>'
strlen(S+,N)
      unifies the second argument, N, with the length of the string, S.
      eg. :- strlen("", N).   --> N = 0
          :- strlen("abc", N).   --> N = 3
substring(String+,Pos+,X-)
      The first argument, String, should be a string, and the second
      argument, Pos, should be a integer. This unifies the third
      argument, X, with the string from the Pos'th character to the
      end of String. The first character's position is 0. When Pos
      is negative, it is treated as the length of String `plus' Pos.
      eg. :- substring("abc",1,X).   --> X = "bc"
          :- substring("abc",-1,X).   --> X = "c"
substrint(String+,Pos+,Length+,X-)}
      The first argument, String, should be a string, and the second
      and third arguments, Pos and Length, should be integers. This
      unifies the fourth argument, X, with a string which is a part of
      String, whose first character is the Pos'th character of String,
      and whose length is Length. The first character's position is 0.
      When Pos is negative, it is treated as the length of String
      `plus' Pos.
      eg. :- substring("abcde",2,2,X). --> X = "cd"
          :- substring("abcde",-2,2,X). --> X = "de"
subsume(X,Y)
        returns true if X is more general than Y.
        eg. :- subsume(X,a).   --> TRUE
            :- subsume(a,X).   --> FALSE
            :- subsume({},{a/b}).   --> TRUE
            :- subsume({a/b},{}).   --> FALSE
sum(X,Y,Z)
      This fails when more than or equal to two arguments are variables.
      This unifies the argument which is a free variable with some

```
      number so that X + Y = Z.
tab
tab(FP+)
      types out the tab code `\t' onto the current output stream
      (the file pointer, FP).
tell(File+)
      sets the current output stream to the file whose path name is
      designated by the argument, File.
talla(File+)
      This is same as tell/1 except for writing at the end of file.
timer(T,C)
      unifies the first argument, T, with the total elapsed time
      (seconds), and the second argument, C, with the time needed
      for constraint transformations since reset_timer is called.
told
      closes the current output stream, and sets it to the screen.
true
      always succeeds.
type(Term+,X)
      unifies the second argument, X, with the type of the first
      argument, Term. The type is among var, integer, float,
      string, file_pointer, pst, clause, list, functor, and atom.
      eg. :- type(X,Y).            --> X = var
          :- type(1,X).            --> X = integer
          :- type(1.02,X).               --> X = float
          :- type("abc",X).              --> X = string
          :- open("programs/test.p","r",T),type(T,X).
                                   --> X = file_pointer
          :- type({a/1},X).              --> X = pst
          :- type((a(U),b(V,W)),X)       --> X = clause
          :- type([a,b,c],X).                  --> X = list
          :- type(a(Y,Z),X).             --> X = functor
          :- type(a,X).            --> X = atom
unbreak
      returns to the break point in step tracing.
unify(C+,NC-)
      unifies the second argument, NC, with the result of
      transforming the constraints in the list C.
var(Term)
      succeeds if Term is a free variable, otherwise fails.
write(Term)
write(Term, FP+)
      writes the term Term onto the current output stream or the file
      pointer FP.
```

5. File I/O

5.1 Loading Programs

In order to load programs from a file, there are two ways:

(a) At the top level, input the name of file enclosed by double quote signs `"', or

(b) provide the file name as the argument for `cup' command.

5.2 Saving Programs

To save a program into a file, there is the following way:

(a) Use CUP command `%w' with file name.

5.3 Input/Output Streams and File Pointers

There are several ways to read from a file or write into a file. One is to use Input/Output Stream. Streams can be set up by see/1 for input, and tell/1 or tella/1 for output. Seen/0 and told/0 are used to close them.

The main defect of using streams is that there should be only one streams for each input and output. Thus the user cannot use several streams at the same time.

Using file pointers will solve this problem. They are set up by open/3, and close/1 is used to close them.

The number of file pointers is limited to 15 for both input and output. The user can check by openfiles/1,/2,/3 what files are open. And she can close them all at once by closefiles/0.

File pointer is represented as a number preceded by #. This can be used as 2nd argument of read/2 and write/2, or 1st argument of nl/1, tab/1.

Example:

```
:- open("programs/test.p",r,X).
--> X = #0       /* #0 is a file pointer */
:- read(X,#0).
--> X = a(1)           /* read the first term of 'test' file */
:- openfiles(X,Y).     /* check open files and their numbers */
--> X = "programs/test.p", Y = #0
```

6. Constraint Transformation

6.1 Usage

The user can use the constraint transformation mechanism of CUP in the following ways:

(1) @ command

One way is to use @ instead of :- at the top level of CUP. For example, by typing as follows,

```
@ member(X,[a,b,c]),member(X,[b,c,d]). [CR]
```

Then, CUP returns equivalent modularly defined constraints and their definitions.

(2) unify/2 predicate

Another way is to use a constraint transformation predicate unify/2. The first argument should be a list or a clause whose members are literals and the second should be a variable as follows:

```
:- unify([c0(X,Y), c1(P,Q,R), c2(Q,S)],Z).
```

unify/2 succeeds if and only if all literals of the first argument can be transformed into modular constraints, and sets the list as the value of the second argument.

(3) Constrained Horn Clause

The user can attach constraints to usual Horn clauses. This is a natural way to use the constraint transformation mechanism in CUP.

6.2 Canonical Form of Constraints

Constraints in Constrained Horn Clauses must be represented in a canonical
form, called `modular' form (However the user can use `%P' command to transform non-modular constraints into modular ones).

Definition 1. modular
A sequence of atomic formulae C1, C2, ..., Cm is modular, if
(1) every argument of Ci is a variable (1 <= i <= m), and
(2) no variable occurs in two different places, and
(3) the predicates occurring in Ci are `modularly defined'.

The predicates used as the constraints in Constrained Horn Clauses may be ordinary Prolog predicates, but they must have the following property.

Definition 2. modularly defined
Predicate p is modularly defined, if and only if the body of the definition of p is either empty or modular.

However the above definition is sometimes too strict. Thus we use the following semimodular constraints when 'M-solvable' mode.

Definition 3. semimodular
A sequence of atomic formulae C1, C2, ..., Cm is semimodular, if
(1) every argument of Ci is a variable (1 <= i <= m), and
(2) no variable occurs in two different places, and
(3) the predicates occurring in Ci are `semimodularly defined'.

Definition 4. semimodularly defined
Predicate p is semimodularly defined, if and only if at least one of the body of the definition of p is either empty or semimodular.

6.3 Operations

Let D, N, and M be initially empty sets. They are used to keep the Derived clauses of generated predicates, Non-modular clauses generated in unfolding, and Modular clauses generated in unfolding, respectively.

Let C be a sequence of formulas which is non-modular, X1,...,Xn are the all variables occurring in C, and p be the atom which is not included in CUProlog database before.

Then we can get the result of modular(C) by the following steps:

1. Add the following definition clause to D and N,
   p(X1,...,Xn) :- C.
2. Repeat the following three operations until N becomes empty:
   (a) unfolding
   Remove one clause z from N, and select an atomic formula A from the body of z. Let a1,...,an be all the clauses in the database of CUProlog whose heads unify with A by the substitution si, and zi be the result of applying s i to z except that A is replaced with the body of ai. Add each zi to M or N depending on whether its body

is modular or not.

In the case that there are no such a, it means that z is unsatisfiable. Then remove all the clauses (recursively) from N and D, which has the head of z in their body.

(b) folding

Remove one clause z (H :- B, C.) from N only in the case that the followings hold:

1. B and C have no variables in common, where B and C are either an atomic formula or a (possibly empty) sequence of formulas.
2. There is a clause P :- Q in D, and there is a substitution s such that Qs = B.

Then add the clause, H :- Ps, C. to N or M, depending on the sequence (Ps, C) is modular or not.

(c) integration

Remove one clause z (H :- B, C.) from N, if the followings hold:

1. B and C ares either an atomic formula or a (possibly empty) sequence of atomic formulas.
2. B and C don't have the same variables in common.
3. B is not modular and contains varibales x1, ..., xn.

Assuming that q be an n-ary predicate symbol which is new to CUProlog database. Then, add H :- q(X1,...,Xn), C. to M or N, depending on whethere the body is modular or not.
And q(Xi,...,Xn) :-B. is also added to D and N.

3. When N becomes empty and M is not, then the transformation of C succeeds. Then we can find the result in M.

Otherwise, the transformation fails, that is, C cannot be transformed into modular form.

The above steps follow the unfold/fold transformation, then the transformations preserve the semantics of the programs.

6.3 Stayin/2

Sometimes people want to define `not equal' in the constraint transformation framework. For example,

```
not_equal(X,Y) :- not eq(X,Y).
```

is assumed to be used in the following way:

```
@ member(X,Some_list), not_equal(X,a).
```

That is, the second argument is always instantiated, and the first argument is usually variable, which comes to be instantiated in some environment. However, if the first argument isn't instantiated when the literal, not_equal, is unfolded, then this must return true, and be deleted from the current constrains.

To avoid it, you can use stayin/2 predicate.
stayin(Condition,PredicateList)
evaluates the Condition argument eachtime the modular mechanism is called.
If it returns TRUE, then the PrdicateList argument is also 'unfolded' (note: the built-in predicates are just evaluated).

Otherwise, the predicate will stay in until the modular mechanism will be called again.
  Thus 'stayin(not var(X),eq(X,a))' can be used to realize the above purpose. You may think this a generalization of 'freeze' mechanism.

6.4 Heuristics in Transformation Process
  CUP needs some control in the transformation process especially the following cases:
      (1) choosing a clause to unfold from N set
      (2) choosing a formula to unfold
CUP uses depth-first heuristics in the former case.
In the later case, MacCup computes the preference (number) to choose the formula to fold:
  preference = 3 * const + 2 * func + vnum - defs + units - 2 * rec + 3 * facts
where
  const = the number of arguments instantiated to atoms
  func = the number of arguments instantiated to complex terms
  vnum = the number of the occurrences of free variables
  defs = the number of definition clauses
  units = the number of unit definition clauses
  rec   = 1 if it has recursive definition, 0 otherwise
  facts = 1 if every definition is unit clause, 0 otherwise
This equation is achieved by experience, but in future it may be replace with
another.

7. Debugger
7.1 Trace Modes
Tracing means to show the status when the spyed predicates are called or exited as true or fail. You can choose either so-called step trace mode or so-called normal trace mode. Furthermore, there are some options to set/remove/show spy points.

      %s   toggle switch for step (interactive) trace. In this mode,
            the prompt becomes `>'.
      %t   toggle switch for normal trace. In this mode, the prompt
            becomes `$'.
      %p * sets spy points on all predicates.
      %p . removes spy points on all predicates.
      %p Pred   toggle switch to set/remove a spy point on Pred
      %p > toggle switch to set/remove a spy point on constraint
            transformations
      %p ? show what predicates are set a spy point

In the step trace mode, each time predicate is called, the execution stops and awaits your command by showing the following prompt:
      #<trace ?>
You can control how to proceed the execution by the following commands:
(This is effective if you set spy points on some appropriate predicates.)

      a   prints the ancestor goals
      b   temporarily exits to the top level. To continue the
            execution, unbreak/0 is used

```
f     makes the current goal fail
h     prints the help message
n     continue the evaluation with normal (non-interactive)
      tracing
s     skips the current goal
x     continue the evaluation without tracing
z     quits refutation
```

7.2 Tracing Constraint Transformations

You can even watch and/or control how the constraint transformation goes by issuing the `%p >' command at the top level and also the trace command (e.g. %s).

If you choose step trace mode, it shows the status and awaits your command, each time the system tries to transform constraints. The following is an example:

```
****DEFS={3}  NON-MODULAR={}  MODULAR={1,2}****
[3(d,0)] c10(V0, V1, V2, V3) <=> one_of([{sem / V0}], {sem / V2}, V3).
[0(g,2)] c9(V0, V1, V2, V3) <=>
                 sc_sl_move([{sem / V0}, {sem / V1}], V2, V3).
<2(m)> c9(V0, V1, Nsc, [{sem / Sl}]) :- c10(V0, V1, Sl, Nsc).
<1(i)> c9(V0, V1, [{sem / V0},{sem / V1}], []).
@step <h,b,q,z,u,s,n,CR>?
```

The first line shows how many predicates to be transformed, and how many clauses are defined as either modular or non-modular. The following lines show the clauses which is to be handled and/or have been defined. The characters shown in the square brackets mean, from left to right, a clause number (e.g. 3) temporarily given, and current status (e.g. d) and the number of its definitions (e.g. 2). The following show what characters are used to represent what kind of status:

```
i     modularly defined unit clause
m     modularly defined clause
g     successfully transformed clause
d     clause to be transformed
u     temporarily defined clause which has non-modular body
```

In the above example, you can find that from the top line, there is only one clause to be transformed whose number is 3, and also there are two modularly defined clauses, whose numbers is 1 and 2. From the following lines, it is shown that the predicate, c10, is going to be transformed, and the predicate, c9, has been transformed and it has two definitions, 1 and 2.

You can control the transformation process by the following commands:

```
[CR]  continue
b     temporarily exits to the top level. To continue the
      execution, unbreak/0 is used
h     prints the help message
n     continue the evaluation with normal(non-interactive) tracing
x     continue the evaluation without tracing
q     abort the process and returns true
```

```
      z     abort the process and returns fail
      u <clause No.> <literal No.>
            specifies which clause is to be unfolded next.

8. Partially Specified Term
      Partially Specified Term, PST for abbreviation, is useful for
representing a structure whose members are attribute-value pairs.
It is useful especially when the number of members cannot be
determined before-hand.
  PST is represented as follows:
      { attribute_1 / value_1, ..., attribute_n / value_n }
where attributes should be atoms and values be terms.  Any two PSTs
are not unifiable only if the values of their corresponding attribute are
not
unifiable. Otherwise they are unifiable.  For example,
 (1) The PST { } is unifiable with any PSTs.
 (2) The PSTs whose attributes are entirely different from each other are
 unifiable.
      eg. {a/1, b/2} and {c/3} are unified into {a/1, b/2, c/3}
 (3) The PSTs which have the same attribute with unifiable values are
 unifiable.
      eg. {a/X, b/2} and {a/1, b/Y, c/3} are unified into {a/1, b/2,
c/3}.
 (4) The PSTs which have the same attribute whose values are not
unifiable
 are NOT unifiable.
      eg. {a/1, b/2} and {a/1, b/3} are not unifiable.

 There are several predicates for dealing with PSTs.
 (1) pvalue(PST+, Pname+, Pvalue)
      unifies the third argument, Pvalue, with the value whose
      attribute is Pname of the partially specified term, PST.
      eg. :- X = {a / 1, b / 2}, pvalue(X, a, Y).   --> Y =    1
          :- X = {a / Y}, pvalue(X, a, c).          --> X = {a / c}
 (2) pnames(PST+, List)
      unifies the second argument, List, with the list whose members
      are the attribute name atoms of the partially specified term, PST.
      eg. :- pnames({a / X, b / c}, Y).       --> Y = [a, b]
 (3) default(Target+, Filter+, DefaultValue+)
      All arguments should be instantiated PSTs. This fails if Target
      doesn't have all attribute-value pairs in Filter. Otherwise,
      Target is unified with a PST whose attribute-value pairs
      are composed from those of DefaultValue which are compatible
      with Target.
      eg. :- X = {a / b, c / d}, default(X, {a / b}, {c / e, e / g}).
      --> X = {a / b, c / d, e / g}
          :- X = {a / b, c / d}, default(X, {a / c}, {e / g}).
      --> fail
 (4) length(X+,N)
  This unifies the second argument, N, with the number of elements in X.
      eg. :- length({a/1, b/2, c/A}, X).--> X = 3
            :- length({}, X).                 --> X = 0

 Note: You can use constrains with PSTs as arguments.  However you get
```

unexpected (incorrect) results because of performance.

9. Syntax of CUP

9.1 BNF Description of CUP

```
------------------------------------------------------------------------
      <char>           ::= <upper> | <lower> | <digit>
      <upper>          ::= A|B|C| ... |X|Y|Z
      <lower>          ::= a|b|c| ... |x|y|z
      <digit>          ::= 0|1|2|3|4|5|6|7|8|9
      <bar>      ::=  |
      <specialchar>    ::= <|>|=|-|+|*|#|:|$|%|&|@|?
      <series>   ::= <digit> | <digit><series>
      <number>   ::= <series> | <series>.<series>
      <string>   ::= <empty> | <char><string>
      <charstring>     ::= "<string>"
      <specialstring>  ::= <specialchar> | <specialchar><specialstring>
      <atom>           ::= <lower><string> | <specialstring> | '<string>'
      <var>      ::= <upper><string> | _<string>
      <constant> ::= <atom> | <number> | <charstring>
      <functor>  ::= <atom>
      <op>       ::= <atom>
      <predicate>::= <atom>
      <list>           ::= [] | [<termList>] | [<termList> <bar> <term>]
      <pstItem>  ::= <atom> / <term>
      <pstItemList>    ::= <pstItem> | <pstItem>, <pstItemList>
      <pst>      ::= { } | { <pstItemList> }
      <term>           ::= <var> | <constant> | <functor>(<termList>) |
                 <list> | <pst> | <op> <term> | <term> <op> <term>
      <termList> ::= <term> | <term>, <termList>
      <literal>  ::= <predicate> | <predicate>(<termList>)
      <bodyLiteral>    ::= <literal> | <var>
      <body>           ::= <bodyLiteral> | <bodyLiteral>, <body>
      <horn>           ::= <literal> | <literal> :- <body> | ?- <body>
      <chc>      ::= <horn>. | <horn> ; <body>.
```

9.2 Built-in operators

```
------------------------------------------------------------------------
      700              xfx             =
      700              xfx             ==
      700              xfx             <=
      700              xfx             <
      700              xfx             >=
      700              xfx             >
      700              xfx             =..
      700              xfy             /
      500              fy              not
(     1000             yfy       ,     )
      1200       xfx             <=>
      1200       yfx             where
      1200       yfx             ;
      1200       fx              ?-
      1200       fx              :-
      1200       xfx             :-
```

Note: comma ',' cannot be redefined.