# Keyed CLP for Hybrid Systems User's Guide

Version 0.90
January 2005

©Kunihiko Hiraishi
School of Information Science,
Japan Advanced Institute of Science and Technology.

# Table of Contents

# 1: Getting Started

## 1-1: Entering Keyed CLP

To enter Keyed CLP interpreter, type 'kclp' at the main operating system prompt:

```
<prompt>kclp
Keyed CLP Interpreter for Hybrid Systems
Version X.X(mm/dd/yyyy)
(C) Kunihiko HIRAISHI, JAIST.

| ?-
```

The prompt '| ?-' indicates that you are at the top level of the Keyed CLP interpreter. You can specify initial consulting files as follows:

```
<prompt>kclp file-name1 ... file-namen
Keyed CLP Version X.X

Consulted : file-name1
...
Consulted : file-namen
| ?-
```

## 1-2: Exiting Keyed CLP

To exit from Keyed CLP interpreter, type ^d (CTRL-d) (or use the system predicate halt) at the top level.

```
| ?- ^d
```

# 2: Loading Programs into Keyed CLP

## 2-1: Loading a File into Keyed CLP

To load a program, use the `consult` predicate:

```
| ?- consult(file-name).
```

or equivalently,

```
| ?- [file-name].
```

If you would like to replace the existing program with a new one, use the `reconsult` predicate:

```
| ?- reconsult(file-name).
```

Adding the following line in a program, Goal is executed when it is loaded.

```
:- Goal.
```

## 2-2: Syntax Error Messages

When a file is loading into Keyed CLP interpreter, Keyed CLP interpreter checks the program for correctness of syntax. If a clause contains a syntax error, Keyed CLP interpreter tells you the point the syntax error has been found.

For a clause shown below,

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

Keyed CLP interpreter displays the following message:

```
syntax error : member(X, [a, b, c, d]. <- here
```

# 3: Running Programs

## 3-1: Asking Questions

Like Prolog, you can ask questions at the top level (each question is called a *goal*).

```
| ?- append([a], [b], Z).
Z = [a, b]

*** yes ***
```

## 3-2: Answers from Keyed CLP Interpreter

When you ask a question to Keyed CLP interpreter, one of three kinds of answers is displayed:

(1)  *** yes ***  with substitution to variables and linear constraints:
There is a solution for the goal.

(2) *** no *** :
There is no solution for the goal.

(3) *** nonlinear *** with linear and nonlinear constraints :
Nonlinear constraints are remained and therefore Keyed CLP interpreter could not answer the question.

```
| ?- X + Y = 1, X - Y = 2.
X = 1.5
Y = -0.5

*** yes ***

0.0100 sec.

| ?- X = 3, Y = 4, X + Y <= 3.

*** no ***

0.0000 sec.

| ?- X = A * B, Y = A - B.
X =  _7
A = ?
B = ?
```

```
Y =  - B + A
_7 = ?

*** nonlinear ***

_7 = A * B

0.0000 sec.

| ?-
```

## 3-3: Backtracking

If there are more than one solutions for a given goal, Keyed CLP displays the following message after displaying the first goal:

```
Retry?
```

If you type 'y', then the system displays the alternatives. If you type 'n', then the execution terminates.

```
| ?- append(X, Y, [a, b, c]).
X = []
Y = [a, b, c]

*** yes ***

0.0100 sec.

Retry? y

X = [a]
Y = [b, c]

*** yes ***

0.0100 sec.

Retry? n

Interrupted.

| ?-
```

## 3-4: Interrupting the Execution of a Program

You can interrupt the execution at any time by typing ^c (CTRL-c).

# 4: Keyed Predicates

## 4-1: Semantics

In addition to ordinary predicates, Keyed CLP has a different type of predicates, called *keyed predicates*, which has the form

Predicate-Name(Key$_1$, ..., Key$_n$ : Arg$_1$, ..., Arg$_m$).

In logic programming languages, each predicate can be seen as a relation. The word key has the same meaning as in the relational database theory, and represents the functional dependency in the relation. Each keyed predicate corresponds to a tuple having Key$_1$,..., Key$_n$ as the key arguments. In a relation, a tuple is uniquely determined if all the values for the key arguments are specified. This property is preserved also in executing programs of Keyed CLP. Every predicate that has the same predicate name and the same values for the keyed arguments must have the same values for all the arguments, i.e., if the values for key arguments Key$_1$,..., Key$_n$ are determined, then Arg$_1$,..., Arg$_m$ must have unique values.

Example 1.

```
p(X : A).
q(X, X + Y):- p(k : X), p(k : Y).

| ?- q(3, Z).
Z = 6.

*** yes ***
```

`p(X : A)` is a keyed predicate and the variable `X` is the key. Execution of Keyed CLP programs is the same as in Prolog excepting that of keyed predicates. In the above program, `q(3, Z)` is evaluated first, and `X = 3` and `Z = X + Y` are extracted as constraints. Next `p(k : 3)` is evaluated, and then the value of predicate p having k as the key value is determined as `p(k : 3)`. Therefore, the next subgoal `p(k : Y)` is unified with `p(k : 3)`, and the value of `Z` is obtained as `3 + 3 = 6`. In this case, this keyed predicate works as a global variable.

The next example shows that keyed predicates can keep variables restricted by some numerical constraints.

Example 2.

```
a(1).
a(2).
p(X : A, B):- A >= B.
q(Y, A1):- p(k : A, Y), a(A1), p(k : A1, _).

| ?- q(2, B).
B = 2

*** yes ***
```

Unifying `p(k : A, 2)` with `p(X : A, B):- A >= B`, the value of predicate p with key value k is determined as `p(k : A, 2)` with a numerical constraint A >= 2. The subgoal `a(A1)` first gets a value `A1 = 1`, and then `p(k : 1, _)` fails because `1 >= 2` is not true. Next `a(A1)` gets a value `A1 = 2`, and then `p(k : 2, _)` succeeds because `2 >= 1` is true. Therefore only `B = 2` is the solution.

## 4-2: Temp Stack

Once a keyed predicate is evaluated, the result is kept in Temp Stack. By system predicate `disp_temp_stack`, the system shows the current contents of Temp Stack.

Let us consider the following program, which calculates the fibonacci sequence.

```
fib(0 : 1).
fib(1 : 1).
fib(N : X1 + X2) :- N > 1, fib(N - 1 : X1), fib(N - 2 : X2).

| ?- fib(3 : X), disp_temp_stack.
Temp Stack:
fib(0 : 1).
fib(1 : 1).
fib(2 : 1 + 1) :- 2 > 1, fib(1 : 1), fib(0 : 1).
fib(3 : 2 + 1) :- 3 > 1, fib(2 : 2), fib(1 : 1).

X = 3

*** yes ***
```

In the execution of `fib(3:X)`, four keyed predicates `fib(0:1)`, `fib(1:1)`, `fib(2:2)`, `fib(3:3)` are added to Temp Stack.

## 4-3: Searching Temp Stack

When a keyed predicate is evaluated, all values for the key part must be fixed. To search predicates in Temp Stack, system predicate *lookup* is provided. It tries to unify the argument with each predicate in Temp Stack.

```
a(X : Y).
|-? a(1 : a), a(2 : b), a(3 : c), lookup(a(_: X)).
X = c

*** yes ***

Retry? y

X = b

*** yes ***

Retry? y

X = c

*** yes ***
```

## 4-4: Special Constant $

'$' is a special constant such that '$ = $' is always false. This is used for storing arbitrary many values without specifying a key.

```
a(X : Y).
|-? a($ : a), a($ : b), a($ : c), disp_temp_stack.
Temp Stack:
a($ : c).
a($ : b).
```

```
a($ : a).

*** yes ***
```

# 5: Debugging Programs

## 5-1: Entering Trace Mode

If you type 'trace' at the top level, the system enters into the trace mode.

```
| ?- trace.

*** yes ***

(trace) | ?-
```

To exit the trace mode, type 'notrace'.

```
(trace) | ?- notrace.
```

## 5-2: Tracing a Program

When the system is in the trace mode, the execution of a program stops at every step each clause is called. The following four commands are available when the system displays the prompt '?' :

(only 'Return' key) : go to the next clause.
`t`   : trace on.
s : skip this clause.
n : trace off.
a : abort the execution.

```
(trace)| ?- append([a], [b], Z).
1-0) CALL : append([a], [b], Z) ?
1-0) TRY  : append([a], [b], [a | _5]) :- append([], [b], _5)
1-0) SUC  : append([a], [b], [a | _5]) :- append([], [b], _5)
1-1) CALL : append([], [b], _5) ?
1-1) TRY  : append([], [b], [b])
1-1) SUC  : append([], [b], [b])
Z = [a, b]

*** yes ***
```

The followings are messages displayed in the trace mode.

| | |
|---|---|
| `>> Tmp` | : a new keyed predicate is added into the temp stack. |
| `>> LIN(n)` | : a new linear constraint is stacked as the n-th constraints. |
| `>> NLIN(n)` | : a new non-linear constraint is stacked as the n-th constraints. |

Traces for predicates begin with letter '_' are not displayed.

## 5-3: Spy Points

You can set (unset) a spy point on a predicate by using the system predicate spy (nospy). The execution of the program stops whenever each predicate with a spy point is evaluated. Spy points work in the non-trace mode.

# 6: Syntax

The following is a BNF grammar for Keyed CLP language.

```
program_instance   :=   clause |   query ;
program   :=   rule .   |   predicate . ;
query   :=   body . ;
rule   :=   functor :- body ;
body   :=   body_atom
        |   body , body
        |   body ; body
        |   ( body ) ;
term : functor   |   list   |   expr   |   string   |   ( body ) ;
list   :=   []
        |   [ tuple ]
        |   [ tuple | variable ] ;
functor   :=   functor_name( tuple : tuple )
        |   functor_name( : tuple )
        |   functor_name( tuple )
        |   fnctor_name ;
tuple   :=   term
        |   term , tuple ;
body_atom   :=   predicate   |   variable   |   constraint ;
constraint   :=   expr = expr
        |   expr > expr
        |   expr >= expr
        |   expr < expr
        |   expr <= expr ;
expr   :=   variable
        |   sysfunc
        |   constant
        |   $
        |   expr + expr
        |   expr - expr
        |   expr * expr
        |   expr / expr
        |   - expr
        |   ( expr );

variable   :=   a string start with a capital letter or _.
```

# 7: Built-in Predicates and Functions

## 7-1: Built-in Predicates

**all(G)** : find all alternatives for goal G.
**answer** : output the answer.
**arg(N, T, A)** : the Nth argument of term T is A.
**atom(T)** : term T is an atom.
**atomic(T)** : term T is an atom or a number.
**avg(E, G, R)** : variable R is the average value of expression E in all alternatives satisfying goal G.
**bagof(T, G, L)** : all alternatives of term T for goal G is the list L.
**call(G)** : execute G.
**clause(H, B)** : there is a clause with head H and body B.
**clean** : clean up the memory space.
**concat(X, Y, Z)** : concatenate strings X and Y giving Z.
**consult(F)** : loading file F.
**copy(T, V)** : make a copy of term T and bind it to V, where all variables in T are replaced by new variables.
**count(G, R)** : variable R is the number of all alternatives satisfying the goal G.
**delete(C)** : delete the program of predicate C.
**disp_temp_stack** : display the keyed predicate stack.
**dump(L)** : display all the constraints related to variables in the list L.
**edit(F)** : edit and reconsult file F.
**fourier(L, R), fourier(L, R, F)** : project the constraints to variables in L and the result is bound to R. If F = 0, then the result is not minimized (but it requires less time).
**list, list(C)** : display the program list (of predicate C). Predicates begins with letter '_' are not displayed.
**lmax(E)** : solve the linear programming problem with a linear objective function maximizing E. It fails if E is unbounded.
**lmax(E, V)**: solve the linear programming problem and set the optimal value of E to V. If E is unbounded then V is bound to 'unbounded'.
**lmin(E)**: solve the linear programming problem with a linear objective function minimizing E. The solution is bound to variables. It fails if E is unbounded.
**lmin(E, V)**: solve the linear programming problem and set the optimal value of E to V. The solution is bound to variables. If E is unbounded then V is bound to 'unbounded'.

**lookup(G)** : execute goal G only in the temp stack (If there are no clauses that matches with G in the temp stack, it fails).
**fail** : backtrack immediately.
**functor(T, F, N)** : term T has name F and arity N.
**funstr(F, S)** : conversion between a functor F and a string S.
**halt** : exit the system.
**max(E, G)** : set the maximum value of E satisfying goal G to E. It fails if G fails or E is unbounded.
**max(E, G, V)** : set the maximum value of E satisfying goal G to V. If G fails then V is bound to 'infeasible'. If E is unbounded then V is bound to 'unbounded'.
**min(E, G)** : set the minimum value of E satisfying goal G to E. It fails if G fails or E is unbounded.
**min(E, G, V)** : set the minimum value of E satisfying goal G to V. If G fails then V is bound to 'infeasible'. If E is unbounded then V is bound to 'unbounded'.
**nl** : send new line code to the current output stream.
**nospy(P)** : remove the spy point on the predicates specified by the predicate name P.
**notrace** : exit from trace mode.
**number(T)** : term T is a number.
**polka_XXX** : polka functions.
**qmax(E), qmax(E, V)**: solve the quadratic programming problem with a quadratic objective function maximizing E (and set the optimal value of E to V). The solution is bound to variables.
**qmin(E), qmin(E, V)**: solve the quadratic programming problem with a quadratic objective function minimizing E (and set the optimal value of E to V). The solution is bound to variables.
**qmax_list(E), qmax_list(E, V)**: same as qmax() except that the objective function is $X_1^2 + X_2^2 + \dots X_n^2$ where E is $[X_1, X_2, \dots, X_n]$.
**qmin_list(E), qmin_list(E, V)**: same as qmin() except that the objective function is $X_1^2 + X_2^2 + \dots X_n^2$ where E is $[X_1, X_2, \dots, X_n]$.
**range(E, G, VMin, VMax)** : set the list of the minimum (maximum) value of each variable in list E satisfying goal G to VMin (VMax). If G fails then VMin and VMax are bound to 'infeasible'. Minimum and maximum values may contain 'unbounded'
**reflist(P, L)** : L is the set of all predicate names which are called by the predicate with predicate name P .
**sum(E, G, R)** : unify variable R with the summation of expression E for all alternatives satisfying goal G.

**see(F)** : make file F the current input stream.
**seen** : close the current input stream.
**spy(P)** : set a spy point on the predicates specified by the predicate name P.
**string(T)** : term T is a string.
**system(S)** : use a shell command.
**tab** : send tab code to the current output stream.
**tell(F)** : make file F the current output stream.
**told** : close the current output stream.
**trace** : enter into trace mode.
**true** : succeed.
**reconsult(F)** : replace the existing program with file F.
**read(T)** : read term T from the current input stream.
**unify(T1, T1)** : unify term T1 with term T2.
**univ(T, L)** : the functor and arguments of term T comprises the list L.
**var(T)** : term T is a variable.
**write(T)** : write term T on the current output stream.

## 7-2: Built-in Functions

@**abs(X)** : absolute value of X.
@**exp(X)** : $e^X$.
@**int(X)** : the maximum integer which is not greater than X.
@**ln(X)** : $\log_e X$.
@**log(X)** : $\log_{10} X$.
@**pi** : $\pi$.
@**rand** : random number.
@**round(X)** : round X.
@**sign(X)** : 1 if X > 0, o if X = 0, -1 if X < 0.
@**sqrt(X)** : the square root of X.
@**atan(X)** : $\tan^{-1}(X)$.
@**cos(X)** : cos(X).
@**sin(X)** : sin(X).
@**tan(X)** : tan(X).
@**pow(X, Y)** : $X^Y$.
@**hash(X, Y)** : X % Y.

# 8: Error Messages

`memory allocation error :` stack-name
The system could not get more memory.

`(system) bad arguments :` clause
An error occurred in the arguments of the system predicate ( e.g. number of arguments, some arguments must be a variable ).

`key :` clause
Key attribute contains some variable(s).

`unknown`
unknown predicate

`(function)  bad arguments :` clause
An error occurred in the arguments of the function.

`must be a number :` clause
An atom or a string is contained in the arithmetic formula.

`system : clause`
An error has occurred in a system predicate.

`file open :` clause
The specified file does not exist.

`max iteration`
The iteration count of the simplex method exceeded the limit.

`nonlinear in optimization`
Constraints contain nonlinear terms when an optimization process has started.

`making tabular`
The size of the simplex tabular exceeds the limit.

`undefined for the key : clause`
There is no clause with the specified key value.