CS-PROLOG

Version 3.25.

MONOPROCESSOR MODEL

CONTINUOUS EXTENSION

DOS VERSION

MULTILOGIC COMPUTING Ltd

BUDAPEST HUNGARY

1. Content

2. Introduction

At the very base CS-PROLOG is a standard PROLOG interpreter/compiler. The compiler uses an extended WAM.

The CS-PROLOG system has two different models for the execution of parallel programs depending on the number of processors at the current machine on which it runs:

monoprocessor model

multiprocessor model

In both models the interpreter/compiler executes different goals simultaneously. To each goal a so called process is assigned. The process is represented by the current path in the search tree underlaying to the goal. The synchronization of the simultaneously working processes is done by messages. The processes can be suspended waiting for messages and these messages are the recommended way of communication between processes. No communication through common logical variables or by modification of the common database is supported. Processes can be generated or deleted dynamically during runtime. Creation and deletion of processes as well as communication is ensured by special built-in predicates **"new", "delete_process", "send", "wait_for"** etc.

Used in a monoprocessor environment the execution of the processes is controlled by an internal scheduler (quasi parallel execution). Process exchange is only possible at certain points of a process (**"wait_for", "hold"**).

Used in a multiprocessor environment it is possible to launch processes on different processors and execute them in parallel. It is possible to have conceptually more processes in the system than processors. In this case on each processor where more then one process is executed the internal scheduler shares the processor between processes the same way as it would do in the monoprocessor case.

Backtracking is supported even in a multiprocessor environment and completeness is ensured by the distributed backtracking algorithm.

However the forward execution is mostly parallel, backtracking is generally sequential. The philosophy of CS-PROLOG is similar to Hoare's CSP this is while CS-PROLOG is standing for the abbreviation of **Communicating Sequential PROLOG.**

Beside of the notion of processes and messages the notion of simulation time is also introduced in CS-PROLOG. Each process evolutes in each own local time. It is possible to assign time duration to the execution of subgoals by mean of special built-in predicates advance and hold. A local clock ticks the elapsed simulation time for each process. Note that the simulation time has nothing to do with the execution times

of programs written in CS-PROLOG. The simulation time is used to model time durations in real systems. Time provides another synchronization mechanism for processes.

In traditional monoprocessor simulation systems the time is unique that is all local clocks at any moment show the same global time.

In a multiprocessor environment maintaining the same notion of global time causes a bottle-neck for the parallelism. Two methods are known to solve this bottle-neck: the so called conservative and the so called optimistic approach (Time Warp). The current version of CS-PROLOG supports the conservative approach. The future versions of CS-PROLOG will support both of them.

When the time changes in discrete steps (may be not uniformly) and events occur in discrete time moments we speak about discrete simulation. If time changes smoothly in a continuous way we are speaking about continuous simulation. Continuous simulation models are generally expressed with the help of differential equations.

In CS-PROLOG versions up to 3.2 it is possible to write only discrete simulation models. Versions 3.3 or higher will support both discrete and continuous simulation modelling and will admit even the combination of them into a so called combined simulation model. Special built-in predicates and clauses serve to support this kind of knowledge based modelling where discrete and continuous components communicate through messages.

This documentation deals with the monoprocessor model under DOS operating system.

The CS-PROLOG system consists of two components:

- The CS-PROLOG interpreter with enhanced program developing environment which enables you to write, modify, run, test and debug your CS-PROLOG program. This component serves for the program development.

- The CS-PROLOG compiler with a byte-code interpreter. Once you have finished the program development you can compile your program into a special format (abstract code for the CS-PROLOG's byte code interpreter) and you can run your application as a stand-alone program. The execution speed is much higher for compiled programs. Except some restrictions (detailed later) the CS-PROLOG language is portable from interpreter to compiler without any modification and the built-in predicate set is fully compatible.

3. The CS-PROLOG Language

3.1 Syntax Of CS-PROLOG

The syntax is that of DEC10-PROLOG except:

- alternatives and groups in a clause are not supported
- the built-in operator set is different
- the operator description differs
- comments until the end of the line are marked with the **"%"** sign

3.2 Language Restrictions For CS-PROLOG Compiler

Compilation of PROLOG programs required the introduction of several changes and restrictions in the language itself. The clauses compiled by the compiler are called in this description 'static', the clauses added runtime by **"add_clause"** are 'dynamic'. The calls in the static program that have no matching static clauses (with the same name and arity) and are not calls of a built-in predicate are considered dynamic. (So the compiler can not signal undefined calls).

It is not allowed to have static and dynamic clauses with the same name (even with different arities). E.g. if in the program there is a clause

help(a,b,c).

you can not add dynamically a clause

help(e,f).

If such a clause is called statically it causes a compile time error.

The syntax of float numbers has changed. At least one digit of the fractional part has to be given. E.g **"1."** is considered as the integer **1** and the period symbol. So the uncomfortable spaces can be avoided in clauses like

```
d(N,M):-  M is N - 1.
```

In the interpreter version as the char sequence **"1."** is interpreted as a float number a space has to be inserted before the period symbol:

```
d(N,M):-  M is N - 1 .
```

3.3 Additional Possibilities

It is possible to generate code statically for dynamic predicates, i.e. to compile partitions of clauses that will be modified dynamically during the execution. By default the clauses in the source code are considered to be static. Using the following pragma

```
dynamic_on.
```

the compiler will generate the code of subsequent clauses as dynamic clauses. The original state can be reset using

```
dynamic_off.
```

The user can increase the efficiency of the code giving more information about a partition with a **"mode"** declaration. In this declaration the types of arguments of a partition are specified:

"+" (in) the argument is always bound when executing

"-" (out) the argument is always an unbound variable when executing

"?" (unknown) we don't know anything about the type.

The **"mode"** declaration has the following form:

```
mode pred_name(in_out_sign1, in_out_sign2, ...).
```

Here **"pred_name"** is the name of the partition.There are as many arguments as the arity of the partition and **"in_out_sign"**-s are **"+"**, **"-"** or **"?"** to specify **"in"**, **"out"** or **"unknown"** types respectively.

For example the classical naive reverse program can be improved using modes:

```
mode naive_reverse(+,-).

naive_reverse([A | X],Z):-
   naive_reverse(X,Y), append(Y,[A],Z).

naive_reverse([],[]).

mode append(+,+,-).

append([A | X],Y,[A | Z]):-
   append(X,Y,Z).

append([],L,L).
```

4. Built-in Predicates

In the description of built-in predicates we use different letters to indicate the different types of arguments. Calling a built-in predicate with an argument type other than indicated here will cause a run-time error. Multiple letters mean a choice of several argument types. The letters listed below refer to the following argument types:

I	non negative fixed point number (e.g.**0, 10**)
N	number (fixed or floating point number, e.g. **0, -0.4**)
C	constant (identifier or string, e.g. **jon** or **"Helen"**)
L	list (e.g. **[1, X, y]**)
V	unbound variable (e.g. **X**)
S	simple value (not variable, list, or compound term)
W	window identifier (e.g. **window_1**)
F	file identifier (e.g.**"file_1"**)
A	value identifier (e.g. **var_1**)
E	arithmetic expression (e.g. **A + B**)
X	any of the above mentioned

If there is more than one argument of the same type than they are indexed. In the following descriptions the phrase "unifies it with **X**" means that if the unification fails the call will fail as well.

A built-in predicate can be

- deterministic or non-deterministic
- backtrackable or non-backtrackable

Deterministic means that during backtracking no new alternative is tried.

Non-deterministic means that during backtracking a new alternative is tried, if exists.

Backtrackable means that during backtracking all global changes are restored to their previous state (undo).

Non-backtrackable means that during backtracking the global changes are not restored.

Without explicit declaration predicates are considered deterministic and non-backtrackable.

4.1 Input-Output Predicates

read(X)
read(WF,X)

Reads the next syntactically correct term either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X.**

read_token(X)
read_token(WF,X)

Reads the next syntactically correct token either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X.**

read_symb(X1,X2)
read_symb(WF,X1,X2)

Reads the next syntactically correct term either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X1.** Unlike **"read" "read_symb"** unifies **X2** with a so called symbolic variable dictionary. Since PROLOG variable names (identifiers beginning with an uppercase letter or underscore) are substituted during reading with a system generated identifier (an underscore followed by a number) the user never can obtain the original symbolic form of a variable. However the user may want to preserve the symbolic form of the variables in the term read in order to make them appear on the output in the original symbolic form. The symbolic variable dictionary unified with **X2** connects variables with their symbolic form. The dictionary is a PROLOG list and contains one list element for each different variable in the term read. For terms containing no variables the symbolic variable dictionary is an empty list. Every item has the following form:

[_nnn | "NAME"]

where **_nnn** is the system generated identifier of the variable **NAME.** Note the variable names like **_nnn** identify the same logical variable. So the user should handle the read term and its symbolic variable dictionary together.

E.g. it is advisable to compose a new term taking both of them and adding the new term to the PROLOG's database by "**add_clause**". Example: reading the term

a(B,C,D,C)

"**read_symb**" the symbolic variable dictionary will be similar to

[[_123 | "B"],[_126 | "C"],[_129 | "D"]]

write(X)
write(WF,X)

Write the contents of **X** either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively.

nl
nl(WF)

Write a newline either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively.

writeq(X)
writeq(WF,X)

Write the contents of **X** either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively. (**q** is for quoted) Unlike "**write**" "**writeq**" writes out a string argument with quotes if necessary. Example:

writeq("A B")

gives

"A B"

on the screen while

write("A B")

gives

A B

on screen.

write_inside(X)
write_inside(WF,X)

Write the contents of **X** either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively. Unlike **"write"** **"write_inside"** omits the parentheses and commas of the outermost level if the value of **X** is a list. Example:

write_inside([a,b])

gives

a b

on the screen.

display(X)
display(WF,X)

Write the contents of **X** either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively. The written term will be the same as in the case of **"write"** but operator expressions (if any) will appear in regular term format. Example:

display(a + b * c)

gives

+(a,*(b,c))

on the screen.

write_symb(X1,X2)
write_symb(WF,X1,X2)

Write the contents of **X1** either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively. Unlike **"write"** **"write_symb"** expects in its **X2** argument a symbolic variable dictionary (see **"read_symb"**). It writes out the **X1** term in a such a way that if a variable has an item in the dictionary then the appropriate symbolic form is written instead of the system generated variable name.

write_spaces(I)
write_spaces(WF,I)

Write a number of **I** spaces either to the default output channel or to the channel identified by **W** (window) or **F** (file) respectively.

open_file(C,CV)

Opens a file for reading. The first argument must be a valid file name in the current operating system. If the second argument is a constant it becomes the inner identifier of that file; otherwise the variable **V** is matched with a file

identifier supplied by the system. It fails if the file could not be opened.

create_file(C,CV)

Opens a file for writing. The first argument must be a valid file name in the current operating system. If the second argument is a constant it becomes the inner identifier of the file; otherwise the variable **V** is matched with a file identifier supplied by the system. It fails if the file could not be created. If the file already exists it will be overwritten.

append_file(C,CV)

Opens a file for appending. The first argument must be a valid file name in the current operating system. If the second argument is a constant it becomes the inner identifier of the file; otherwise the variable **V** is matched with a file identifier supplied by the system. It fails if the file could not be found nor created. If the file already exists the next output requests will be appended to the end of the file. If the file don't exist yet a new file will be created.

close_file(F)

Closes a file identified by **F**.

set_input(WF)
set_input(WF,X)

Sets the default input channel either to the channel identified by **W** (window) or **F** (file) respectively. If a second argument is given the previous default channel identifier is unified with it.

set_output(WF)
set_output(WF,X)

Sets the default output channel either to the channel identified by **W** (window) or **F** (file) respectively. If the second argument is given the previous default channel identifier is unified with it.

get_input(X)

Unifies the current input channel identifier with **X**.

get_output(X)

Unifies the current output channel identifier with **X**.

read_from_string(C,X)
read_from_string(C,X,X1)

Reads a syntactically correct object from the constant **C** and unifies it with **X**. **C** does not necessarily have to contain one single PROLOG term. If **C** contains more than one term the

first term is read. If **X1** is given the remainder string is unified with it.

read_from_string_symb(C,X1,X2)

Reads a syntactically correct object from the constant **C** and unifies it with **X1**. It unifies **X2** with the symbolic variable dictionary. (See **"read_symb"**.)

write_to_string(V,X)

Forms a string representing the content of **X** exactly as **"write"** does and unifies it with **V**.

write_to_string_symb(V,X1,X2)

Forms a string representing the content of **X1** exactly as **"write_to_string"** does and unifies it with **V**. Unlike **"write_to_string"** **"write_to_string_symb"** expects in its **X2** argument a symbolic variable dictionary (see **"read_symb"**). It forms the **X1** term in a such a way that if a variable has an item in the dictionary then the appropriate symbolic form is used instead of the system generated variable name.

write_inside_to_string(V,X)

Same as **"write_to_string"** except that **"write_inside_to_string"** omits the parentheses of the outermost level and all the commas in that list if the value of **X** is a list.

get0(X)
get0(WF,X)

Reads the next byte either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X**.

get(X)
get(WF,X)

Reads the next printable character either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X**. Printable characters are the ascii code of which is greater then 32.

read_record(X)
read_record(WF,X)

Reads the next character record either from the default input channel or from the channel identified by **W** (window) or **F** (file) respectively and unifies it with **X**. A character record consists of all the characters between the current character of the input stream and the next newline character. The newline character itself won't belong to the character record but will be removed from the input stream.

4.2 Database Handling Predicates

NOTICE: (for database handling predicates) If an argument which is a clause contains an operator with negative priority (e.g. ":-", ",") then it must be enclosed in parentheses!

add_clause(X)
add_clause(X,I)

X must be a clause. Enters the clause X to CS-PROLOG's database. The new clause is either appended to the end of the partition of X or inserted as the **I**th clause of the partition. A partition is an ordered set of clauses that have the same name. If **I** is equal zero or greater than the number of clauses already in the partition then the clause is appended.

delete_clause(C,I)

Deletes the **I**th clause of the partition named **C** from CS-PROLOG's database. If the clause can not be found "**delete_clause**" sends error message.

delete_partition(C)

Deletes all clauses of the partition **C**.

get_clause(C,I,X)

Unifies the **I**th clause of the partition **C** with **X**. If the clause can not be found "**get_clause**" fails.

find_clause(X1)
find_clause(X1,X2)

X1 must be a partially qualified clause with the name of the clause given. "**find_clause**" unifies **X1** with the first clause in CS-PROLOG's database that matches it. Every time when "**find_clause**" is executed again during backtrack it unifies **X1** with the subsequent matchable clause in a non-deterministic way. "**find_clause**" fails if no more such clause exists. If **X2** is given it is unified with the serial number of the clause in its partition (This predicate is non-deterministic and non-backtrackable).

clause_count(C,X)

Unifies the number of clauses of the partition **C** with **X**.

assert_clause(X)
assert_clause(X,I)

Performs the same task as the predicate "**add_clause**". However when backtracking reaches this point the asserted clause will be removed from CS-PROLOG's database. (This predicate is deterministic and backtrackable).

suppress_clause(C,I)

Performs the same task as the predicate **"delete_clause"**. When backtracking reaches this predicate the suppressed clause will be reloaded to CS-PROLOG's database with the same serial number as it had when it was suppressed. (This predicate is deterministic and backtrackable).

suppress_partition(C)

Performs the same task as the predicate **"delete_partition"**. When backtracking reaches this predicate all suppressed clauses will be reloaded to CS-PROLOG's database. WARNING: Intermixing backtrackable and non-backtrackable data-base handling predicates for the same partition can lead to unexpected results. (This predicate is deterministic and backtrackable).

set_value(C,S)

Stores the simple expression **S** (anything that is not a list or a compound term) as value of the global variable named **C**. The previous value of the variable is overwritten. The constant **C** becomes the inner identifier of this global variable.

get_value(A,X)

Returns the value of the global variable **A** by unifying it with **X**.

incr_value(A,X)

If the value of the global variable is a number then **"incr_value"** increments it by one and returns the result unifying it with **X**. If the value is not a number an error occurs.

set_value_b(C,S)

Performs the same task as the predicate **"set_value"**. However when backtracking reaches this point the variable will be reset to its previous value. (This predicate is deterministic and backtrackable).

incr_value_b(A,X)

Performs the same task as the predicate **"incr_value"**. However when backtracking reaches this point the variable set will be reset to its previous value. (This predicate is deterministic and backtrackable).

comp(L,X)

Composes a new term from the elements of **L** and unifies it with **X**. The first element in **L** will be the name of the new term. The remaining elements will form the arguments of the new term in the same order as in **L**.

decomp(X1,X2)

X1 must be a term. "**decomp**" decomposes **X1** to a list and unifies it with **X2**. Its name will be the first element of the list and its arguments will be the remaining elements of the list in the same order as in the term.

save_partition(F,C)

C must be the name of a partition in the PROLOG's database. "**save_partition**" stores the named partition to the file **F** in text format. If the partition cannot be found error message is sent.

load_file(C)

C must be a filename in the current operating system. The file should contain syntactically correct PROLOG clauses. "**load_file**" reads all clauses and adds them to the PROLOG's database as "**add_clause**" would do. If the file cannot be found it fails. If syntactical error occurs then reading is abandoned and error message is sent. But clauses previously read and added remain in the database.

local_add_clause(X)
local_add_clause(X,I)
local_delete_clause(C,I)
local_delete_partition(C)
local_get_clause(C,I,X)
local_find_clause(X1)
local_find_clause(X1,X2)
local_clause_count(C,X)
local_assert_clause(X)
local_assert_clause(X,I)
local_suppress_clause(C,I)
local_suppress_partition(C)
local_set_value(C,S)
local_get_value(A,X)
local_incr_value(A,X)
local_set_value_b(C,S)
local_incr_value_b(A,X)

The scope of the previously described database handling predicates is global, i.e. changes in the PROLOG's database made by one of them in one process is visible in another process. Predicates with the "**local_**" prefix do the same task as their global counterpart but their scope is local to the calling process, i.e. changes in the local database of a process is invisible for every other process. The intermixed use global and local database handling predicates inside a process isn't allowed and the effect is undefined.

4.3 Predicates Controlling Execution

succeed

Always succeeds.

fail

Always fails.

eq(X1,X2)

Unifies **X1** with **X2**. "**X1 = X2**" is the same as "**eq(X1,X2)**".

! **cut**
!(X) **cut ancestor**

If "**!**" (cut) is executed it always succeeds. When backtracking reaches its calling point then "**!**" (cut) prohibits every other choice between itself and its parent call.

If "**!(...)**" (cut ancestor) is executed it fails unless it finds an ancestor unifiable with **X**. When backtracking reaches its calling point then "**!(...)**" (cut ancestor) prohibits every other choice between itself and the found ancestor unifiable with **X**. The ancestor to be found must have been invoked through "**ancestorable_call**".

ancestor(X)

Tries to find the youngest ancestor unifiable with **X**. The predicate fails if none is found. The ancestor to be found must have been invoked through "**ancestorable_call**".

ancestorable_call(X)

X must be term representing a PROLOG call. If a call **X** is referenced by an "**ancestor**" or "**!(...)**" (cut ancestor) it has to be invoked through the "**ancestorable_call**" predicate. E.g. "**ancestorable_call(a_call(1,Z))**" invokes "**a_call(1,Z)**" as usual but later this call can be referenced by either of "**ancestor**" or "**!(...)**" (cut ancestor) predicates otherwise no ancestor would be found and both of them would fail.

4.4 Inquiring Predicates

is_num(X)

Succeeds if the argument represents either a fixed or a floating point number otherwise fails.

is_int(X)

Succeeds if the argument represents an integer number otherwise fails.

is_float(X)

Succeeds if the argument represents a floating point number otherwise fails.

is_atom(X)

Succeeds if the argument represents an atom - name with lower case initial letter or a string (anything between double quotes) - otherwise fails.

is_file(X)

Succeeds if the argument represents a file identifier otherwise fails.

is_window(X)

Succeeds if the argument represents a window identifier otherwise fails.

is_value(X)

Succeeds if the argument represents a value identifier otherwise fails.

is_list(X)

Succeeds if the argument represents a list otherwise fails.

is_var(X)

Succeeds if the argument is an unbound variable otherwise fails.

is_ground(X)

Succeeds if the argument is a ground constant variable otherwise fails.

type_of(X1,X2)

Unifies **X2** with one of the constants

int
float
file
window
value
atom
list
var
expr
ground_constant
prolog_pred
nonprolog_pred

corresponding to the type of **X1**.

list_length(L,X)

Unifies the length of the given list **L** with **X**.

4.5 Arithmetics

All arithmetic functions are collected in the built-in predicate "**is**" which appears in infix format:

X is E

E is an arithmetic expression composed of numerical constants, variables, arithmetical built-in operators and parentheses. Numerical constants are either integer or floating point numbers. Variables must have numerical values at the moment of evaluation otherwise an error message is generated. The built-in arithmetic operators are of either unary or binary type.

It is ambiguous to write

X is 1+2.

because it is not clear whether the period is a decimal point or the end mark of the term. Also writing

X is 1-2.

is ambiguous because minus is considered to be the sign and not an operator. To solve these problems leave spaces

between number and period and between numbers and operators:

```
X is 1 + 2 .
X is 1 - 2 .
```

The binary operators are:

E1 + E2	**add**
E1 - E2	**subtract**
E1 * E2	**multiply**
E1 / E2	**divide**
E1 mod E2	**modulo**
E1 ^ E2	**power**

If either operand of the binary operators has a floating point value the result will be a floating point number otherwise an integer number. Exception: the power operator always generates a floating point number.

The unary operators are:

+E	**unary plus**
-E	**unary minus**
abs(E)	**absolute value**
sqrt(E)	**square root**
exp(E)	**exponential function**
log(E)	**natural logarithm**
sin(E)	**sine**
cos(E)	**cosine**
tan(E)	**tangent**
asin(E)	**arc sine**
acos(E)	**arc cosine**
atan(E)	**arc tangent**
floor(E)	**truncate decimal part**
trunc(E)	**truncate decimal part**

These predicates always return floating point values. Exceptions: "**unary plus**" and "**unary minus**" return the same type as their argument. "**trunc**" returns fixed point value. The trigonometrical functions expect their argument to be in

radian and the inverse trigonometrical functions return their values in radian.

4.6 Comparison Predicates

N1 < N2	**numerically less**
N1 <= N2	**numerically less or equal**
N1 > N2	**numerically greater**
N1 >= N2	**numerically greater or equal**
N1 =:= N2	**numerically identical**
N1 =\= N2	**numerically non-identical**
C1 @< C2	**lexicographically less**
C1 @<= C2	**lexicographically less or equal**
C1 @> C2	**lexicographically greater**
C1 @>= C2	**lexicographically greater or equal**

These predicates test whether the appropriate relation is true for the given two arguments. The two arguments must be of the same type; no mixed argument type arguments are allowed. Comparison of constants means lexicographical comparison.

NOTICE: "**X1 = X2**" is a synonym of "**eq**" (see chapter "Predicates Controlling Execution").

4.7 String Manipulating Predicates

string_length(C,X)

Unifies the length of constant **C** with **X**.

concat(C1,C2,X)

Unifies the concatenation of constants **C1** and **C2** with **X**.

substring(C,I,X)
substring(C,I1,I2,X)

Unifies a substring of constant **C** with **X**. This substring begins at the **I**th or **I1**th character of **C** respectively and is of the length **I2** or until the end of **C** respectively.

search_pattern(C1,C2,X)

Searches the first occurrence of constant **C2** within the constant **C1**. If this pattern can be found then its index otherwise zero is unified with **X**.

char_of(I,X)

I must be in the range [1,255]. **"char_of"** unifies **X** with a single character constant of the ASCII code **I**.

code_of(C,X)
code_of(C,I,X)

Unifies **X** with the ASCII code of the first or the **I**th character of **C**.

4.8 Window Handling

4.8.1 Window Basics

The notion of the window is the following. A window is a rectangle on the screen defined by 5 numbers:

- row and column of upper left corner
  (0 - 24, 0 - 78)
- number of rows and number of columns
  (1 - 25, 1 - 80)
- attribute number which defines the foreground and background color and the frame of the window

The colors are the normal IBM DOS color values:

0 - black

1 - blue

2 - green

3 - cyan

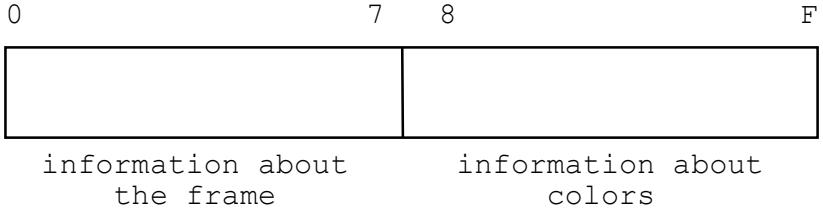
4 - red

5 - magenta

6 - brown

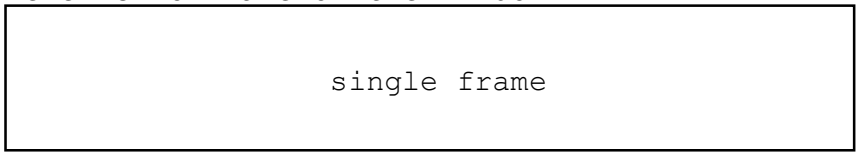
7 - white

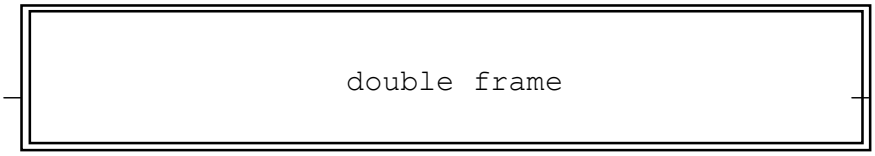
The attribute is an integer. This number is used as a 16 bit pattern:



The left side byte can have any of the following 5 values:

0 There is no frame on the window



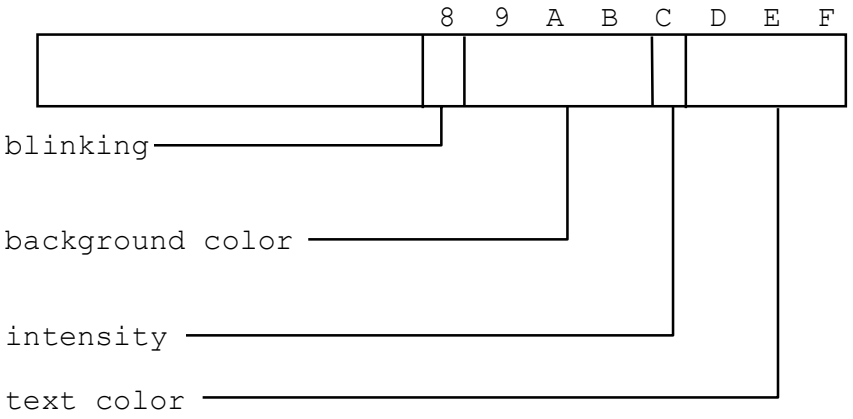
2 

3 

4 

All types of frames need a one character wide border. The size of the window always contains the rows and columns needed for the frame.

The right byte of the attribute is divided into four parts which have the same meaning as the original DOS attribute:



The attribute value can be calculated by the following expression:

```
attr = 256 * frame_type        +
       128 * blink_bit         +
       16  * background_color +
       8   * intensity_bit     +
             text_color.
```

4.8.2 Window Handling Predicates

create_window(I1,I2,I3,I4,I5,CV)

Creates a window. **I1** is the number of upper row, **I2** is the number of left column, **I3** is the number of rows, **I4** is the number of columns, **I5** is the attribute. If the sixth argument is a constant **C** then it becomes the inner identifier of the window otherwise the variable **V** is matched with a window identifier supplied by the system.

get_window(W,X1,X2,X3,X4,X5)

Unifies the last five arguments with the parameters of the window **W** with the same meaning as in **"create_window"**.

delete_window(W)

Deletes the window associated with the name **W.** Window identifier **W** will be an atom. The window on the screen is not affected.

assign_text(W,I1,I2,C)

Assigns the constant **C** to the window **W. I1** and **I2** stand for the relative row and column number of the first character of the assigned text calculated from the upper left corner of the window. When opening window **W** this text will be displayed automatically. The default is the empty string (see also **"read_window"** below).

assign_key(W,I1,I2)

Assigns two "send" keys to the window **W. I1** and **I2** may be identical and must represent the ASCII code of a key. When in the window **W** the system waits for input pressing one of the two assigned keys will send the input, i.e. terminate the input. Defaults are the Enter and Escape keys. This works only with the use of **"read_window"** and menu predicates.

assign_global_key(L)

Assigns send keys to all windows in the system. **L** must be list containing integer numbers of ASCII codes to assign or an empty list. If **L** isn't an empty list **"assign_global_key"** assigns all of them as an additional set of send keys to all windows. The individual send keys provided by the window handler as default or assigned using **"assign_key"** are still working. The additional send key set works until another **"assign_global_key"** call change it. If **L** is empty list then the additional send key set is deleted but the individual send keys are not affected.

open_window(W)

Opens the window **W**. The empty window will appear on the screen with the assigned text if any. This predicate can also be used to clear a previously opened window.

close_window(W)

Closes the window **W** leaving a black rectangle on the screen. Its use is not necessary and has only been retained for reasons of compatibility with the previous versions.

change_color(W,I1)
change_color(W,I1,I2,I3,I4)

Changes the information about colors (right byte of attribute) of the window **W** to **I1** on the entire window or on line **I2** and column **I3** for **I4** character positions.

scroll_window(W,N)

N must be an integer. "**scroll_window**" scrolls the window **W** up if **N** is positive and down if **N** is negative by the absolute value of **N** lines.

write_window(W,I1,I2,C)

Writes the constant **C** to the window **W. I1** and **I2** mean the relative row and column number of the first character of the text calculated from the upper left corner of the window.

read_window(W,I1,I2,I3,X)
read_window(W,I1,I2,I3,X,X1)

Executing this predicate first the cursor appears in the window **W** at relative position (**I1, I2**). The user can then type in text to the window using any of the cursor and editing keys. The input can be finished by pressing one of the "send" keys (see "**assign_keys**" above). Then the argument **X** is unified with a constant which is extracted from the window field beginning at (**I1,I2**) position and length **I3**. **X1** is unified with the ASCII code of the send key used.

read_window_text(W,I1,I2,I3,X)

Performs the same task as "**read_window**". However the cursor does not appear and "**read_window_text**" only reads previously written text from the window.

hor_menu(W,I,L,I1,I2,X)

"**hor_menu**" defines and creates a horizontal menu for comfortable user input. **W** is the window that the menu uses; **I** is the number of the row to be used in the window; **L** is a list of sublists with the column position and the menu item; **I1** is the number of the item to be highlighted first; **I2** is the highlight attribute; **X** is a variable that will be unified with the item selected or with 0 if the Escape key was pressed. If

the menu items contain only one capital letter entering this letter chooses the first item containing that capital letter.

ver_menu(W,I,L,I1,I2,X)

"**ver_menu**" defines and creates a vertical menu for comfortable user input. **W** is the window that the menu uses; **I** is the column to start at within the window; **L** a list with constants as items to be displayed and returned; **I1** is the number of the item to be highlighted first; **I2** is the highlight attribute; **X** is a variable that will be unified with the item selected or with 0 if the Escape key was pressed. If the constants contain only one capital letter entering this letter chooses the first item containing that capital letter. Cursor-left returns -1, cursor-right returns +1 independent of which item was highlighted last.

4.8.3 Screen Levels

It is often necessary to preserve the contents of a screen before opening a new window if you want to get the same state of the screen after closing that new window. This cannot be done using "**close_window**" because "**close_window**" clears part of the contents of the screen. However it is possible to open a new level of the screen and thereafter open the appropriate window. Closing that new screen level you get the same screen state as before. This can be thought of as if one puts a piece of glass on the screen. The text under the piece can be seen but text and windows displayed on it are printed only on that piece of glass. If you return to the previous level it is as if you take away the glass cover and look at the original screen. There are 16 screen levels available one of which is the actual screen level that is displayed. You can arbitrarily switch between them.

set_screen(I)

I must be less than 16. "**set_screen**" makes the **I**th virtual screen appear on the display.

get_screen(X)

Unifies the number of the currently visible screen with **X**.

clear_screen
clear_screen(I)

Clears the currently visible screen on the display if the first form is used or if **I** is the current screen. Otherwise the **I**th virtual screen will be cleared in the memory but the display will not be affected.

copy_screen(I1,I2)

Copies the **I1**th virtual screen to **I2**.

open_level

Opens a new level.

close_level

Closes the most recently opened level.

4.9 System Handling Predicates

save_system(C)

C must be a valid filename in the current operating system. **"save_system"** saves the current state of the CS-PROLOG system to the file specified by **C**. Later that state may be restored with **"load_system"**.

load_system(C)

C must be a valid filename in the current operating system. **"load_system"** loads the state of the CS-PROLOG system saved by **"save_system"**. The current state will be lost!

4.10 Operators

4.10.1 Operator Basics

An operator is characterized by its name, its type and its priority. The type must be one of the following constants:

pf - unary prefix

sf - unary suffix

lr - binary infix left to right

rl - binary infix right to left

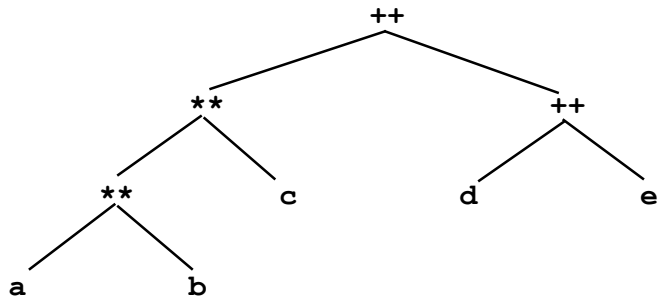
The priority must be an integer in the range between -3000 and 3000.

Example: If ****** is defined as binary infix operator left to right with priority 8 and **++** is defined as binary infix operator right to left with priority 4 then the expression

a ** b ** c ++ d ++ e

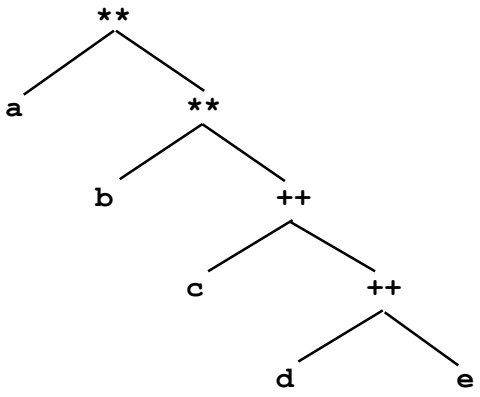
can be illustrated in the following parenthesized and tree format:

```
( ( a ** b ) ** c ) ++ ( d ++ e )
```



If the direction and priority of the ****** are changed to right to left and 2 respectively then the same expression will define a quite different term:

```
a ** ( b ** ( c ++ ( d ++ e ) ) )
```



4.10.2 Predefined Operators

The built-in operators of CS-PROLOG are the following:

Name	Type	Priority
:-	unary prefix	-3000
:-	binary infix left to right	-3000
,	binary infix right to left	-1000
<	binary infix left to right	0
<=	binary infix left to right	0
=	binary infix left to right	0
\=	binary infix left to right	0
>	binary infix left to right	0
>=	binary infix left to right	0

is	binary infix left to right	0
@<	binary infix left to right	0
@<=	binary infix left to right	0
@>	binary infix left to right	0
@>=	binary infix left to right	0
=:=	binary infix left to right	0
=\=	binary infix left to right	0
+	unary prefix	200
+	binary infix left to right	200
-	unary prefix	200
-	binary infix left to right	200
*****	binary infix left to right	300
/	binary infix left to right	300
mod	binary infix left to right	400
^	binary infix right to left	500
:	binary infix left to right	600

4.10.3 Operator Handling Predicates

add_operator(C1,C2,I)

Creates a new operator with the name **C1**. **C2** is the type and **I** is the priority of the new operator. **C2** must be the one of the constants **pf**, **sf**, **lr**, or **rl**. **I** must be in the range between -3000 and 3000.

Operators can be defined statically in the source program as well. A clause "**operator(C1,C2,I)**" has the same effect during the **LOAD**ing, **ENTER**ing or **MODIFY**ing as the predicate "**add_operator(C1,C2,I)**".

get_operator(C,X1,X2)

C must be an operator name. Unifies **X1** with kind (**lr**, **rl**, **sf**, **pf**) and **X2** with the priority of the operator **C**. If **C** is not an operator name it fails. If **C** is defined with multiple kind and **X1** an unbound variable then the operator with the first kind is returned considering the next order: **lr**, **rl**, **sf**, **pf**.

4.11 Graphics

4.11.1 Eagle Graphics

The CS-PROLOG graphics, called "eagle graphics", is similar to the so-called "turtle graphics". It enables the user to draw points, lines, filled areas and geometric figures in a three-dimensional space. The eagle lives in the three dimensional space and its activity can be controlled by

built-in graphic procedures. If the pen of the eagle is put down it leaves its path in the three-dimensional space which is projected onto the two-dimensional display screen. The current state of the eagle and the graphic system is characterized by several system variables.

In eagle geometry there are two coordinating systems: one is the absolute orthogonal system, the other is a relative coordinate system which the eagle carries on his back. The origin of the absolute three-dimensional coordinate system is projected to the middle of the two-dimensional screen in such a way that axis X is horizontal (positive to the right) and axis Y is vertical (positive to the top) while axis Z points into the screen. On the other hand axis X of the eagle's system is the direction in which the eagle is facing. Axis Y can be associated with the direction of the expanded wings of the eagle. Axis Z is orthogonal to the eagle's (X,Y) plane. The eagle can turn around any of its axes thereby moving its own coordinate system.

The CS-PROLOG graphics enables the user to draw with more than one eagle. A maximum of eight eagles may be present on the screen. An eagle may be either awake or asleep. Eagles are numbered from 0 to 7. Thus the eagle number must be in the (0,7) range. Default state is that eagle number zero is awake and all the others are asleep. All graphic predicates except those which refer to the whole graphic system have two forms. The shorter form performs the operation on all eagles awake. In this case the operation is the same for all awake eagles but the current position of each one can be different consequently its effect too. The longer form contains an extra argument to specify the number of the eagle to carry out the operation. In that case the eagle performs the specified operation no matter whether it is awake or asleep. Exception: the predicates **"forward"**, **"backward"**, and **"fly"** do nothing when the specified eagle is asleep.

4.11.2 Eagle Activating Predicates

alive(I)

Succeeds if the **I**th eagle is alive otherwise fails.

wake(I)

Wakes up the **I**th eagle.

sleep(I)

Puts the **I**th eagle to sleep.

sleepall

Puts all eagles to sleep.

4.11.3 Drawing Predicates

forward(N)
forward(N,I)

Moves all awake eagles or the **I**th eagle respectively forward by **N** logical units.

backward(N)
backward(N,I)

Moves all awake eagles or the **I**th eagle respectively backward by **N** logical units.

turn(N)
turn(N,I)

Turns all awake eagles or the **I**th eagle respectively around the Z axis by **N** degrees.

tilt(N)
tilt(N,I)

Turns all awake eagles or the **I**th eagle respectively around the Y axis by **N** degrees.

twist(N)
twist(IN,I)

Turns all awake eagles or the **I**th eagle respectively around the X axis by **N** degrees.

fly(L)
fly(L,I)

L must have exactly three integer elements. "**fly**" moves all eagles awake or the **I**th eagle respectively to the point in the three-dimensional space specified by triplet in **L**. The orientation of the eagles does not change.

look(L)
look(L,I)

L must have exactly three integer elements. Turns all awake eagles or the **I**th eagle respectively in such a way that the eagle(s) will face to the point specified by this triplet. The position of the eagles does not change.

visible
visible(I)

If the projection is perspective and the local (X,Y) plane of the 0th or the **I**th eagle is visible from the viewpoint then "**visible**" succeeds otherwise fails. This predicate is useful for hidden lines removing in polyhedral units.

draw_text(C)
draw_text(C,I)

Writes out the string **C** horizontally on the graphic screen for all eagles awake or for the **I**th eagle respectively. The string begins on the screen near the point which is the projection of the eagle's current three-dimensional position.

fill(I1)
fill(I1,I2)

Fills a closed region around each eagle awake or the **I2**th eagle with the color specified by **I1**.

4.11.4 Global Graphic State Indicators

The following predicates serve to set the global graphic system variables and to get their current values.

graphic(CV)

If the argument is a constant it must be either **"on"** or **"off"**. **"graphic"** switches the screen to graphics mode or to text mode. If its argument is a variable **"graphic"** unifies it with the current screen mode. The default value is **"off"**.

background(IV)

If the argument is an integer it must be in the range between 0 and 15. **"background"** sets the background to be of the color specified by **I**. If the argument is a variable **"background"** unifies the background color with it. The default color is 1.

palette(IV)

If the argument is an integer it must be either 0 or 1. **"palette"** switches between the two palettes of a graphic screen. If the argument is a variable **"palette"** unifies it with the current palette. The default value is 1.

resolution(CV)

If the argument is a constant then it must be either **"low"** or **"high"**. **"resolution"** switches the resolution of the graphic screen to the low or high resolution mode respectively. If the argument is a variable **"resolution"** unifies it with current resolution. The default value is **"low"**.

ration(NV)

If the argument is a number it specifies the quotient of the horizontal and vertical physical measurement of the screen. If the argument is a variable **"ration"** unifies it with the current value of the screen ratio. The default value is 1.

4.11.5 Graphic State Indicators For Eagles

horizont_scale(NV)
horizont_scale(NV,I)

If the first argument is a number it specifies the ratio between the logical and physical width of screen considering all awake eagles or the **I**th eagle respectively. If the first argument is a variable it unifies the current value of the horizontal scale of the 0th or Ith eagle respectively. The default value is a floating point number representing the ratio 320/1024.

pen(CV)
pen(CV,I)

If the first argument is a constant it must be either **"up"** or **"down"**. **"pen"** defines the pen position of all awake eagles or the **I**th eagle respectively. If the first argument is a variable it unifies the pen position of the 0th or **I**th eagle respectively. The default position is **"down"**.

pen_color(I1V)
pen_color(I1V,I2)

If the first argument is an integer it must be in the range between 0 and 3. **"pen_color"** defines the pen color of all awake eagles or the **I2**th eagle respectively. If the first argument is a variable it unifies the pen color of the 0th or **I2**th eagle respectively. The default color on is 1.

projection(CV)
projection(CV,I)

If the first argument is a constant it must be either **"linear"** or **"perspective"**. **"projection"** specifies the projection mode of all eagles awake or the **I**th eagle respectively. If the first argument is a variable it unifies the projection mode of the 0th or **I**th eagle respectively. The default projection is **"perspective"**. In perspective projection you get the 2-D coordinates X, Y from the 3-D coordinates x, y, z as follows:

X = a - b * (x - a) / (z - c) and

Y = b - c * (y - b) / (z - c)

where a, b, c are the coordinates of the viewpoint.

view_point(LV)
view_point(LV,I)

If the first argument is a list this list must contain exactly three integer numbers. This triplet specifies the view point in the three-dimensional space for perspective projection considering all eagles awake or the **I**th eagle respectively. If the first argument is a variable it unifies

the view point of the 0th or **I**th eagle respectively. The default view point is **[0,0,2000]**. The viewing plane is the XY plane of the fixed coordinate system. The further the viewpoint is from the object the bigger the projection becomes.

position(LV)
position(LV,I)

If the first argument is a list this list must contain exactly three integer numbers. This triplet specifies the eagle position in the three-dimensional space considering all eagles awake or the **I**th eagle respectively. If the first argument is a variable it unifies the eagle position of the 0th or **I**th eagle respectively. The default eagle position is **[0,0,0]** which is in the middle of the screen.

head_position(LV)
head_position(LV,I)

If the first argument is a list this list must contain exactly three sublists. Each sublist must contain exactly three integer numbers. This 3 x 3 matrix specifies the eagle orientation in the three-dimensional space considering all the eagles awake or the **I**th eagle respectively. If the first argument is a variable it unifies the eagle orientation of the 0th or **I**th eagle respectively. The default eagle orientation is **[[1,0,0],[0,1,0],[0,0,1]]**. This means alignment of the eagle's axes with the fixed coordinate system.

linear_map(LV)
linear_map(LV,I)

If the first argument is a list this list must contain exactly two sublists. Each sublist must contain exactly three integer numbers. This 2 x 3 matrix describes the mapping of the linear projection considering all eagles awake or the **I**th eagle respectively. If the first argument is a variable it unifies the linear mapping of the 0th or **I**th eagle respectively. The default linear mapping is **[[1,0,0.707], [0,1,0.707]]** which corresponds to front view. You get the 2-D coordinates X, Y from the 3-D coordinates x, y, z as follows:

X = a * x + b * y + c * z and

Y = d * x + e * y + f * z

with the linear map

[[a,b,c],[d,e,f]].

4.11.6 Miscellaneous Graphic Predicates

save_picture(C)

C must be a valid file name in the current operating system. Saves the content of the screen or the graphic screen buffer to the file C depending on if the screen is in graphic or text mode.

load_picture(C)

C must be the filename of a previously saved picture created by **"save_picture"**. If the file cannot be found it fails. Loads the content of the file C to the screen or the graphic screen buffer depending on if the screen is in graphic or text mode.

clear_gr_screen

Deletes the contents of a graphic screen.

init_indicator

Gets the graphic state indicators to the default values.

reset_heading
reset_heading(I)

Sets the head orientation of all awake eagles or the **I**th eagle respectively to the default values.

hard_copy

Prints the contents of the graphic screen on the printer.

4.12 Miscellaneous System Predicates

sound(N1,N2)

Beeps in **N1** Hertz frequency for **N2** milliseconds.

color_mode

Succeeds if the system is in color mode and fails if it is in black-white mode.

cpu_time(X)

Unifies **X** with the cpu time measured in milliseconds returned by DOS.

datetime(X)

Unifies **X** with a list containing the actual year (current year minus 1900), month (0-11, January = 0), day in the year (0-365 jan 1 = 0), day in the month (1- 31), day in the week (0-6 Sunday=0), hour (0-24), minute, second.

pause

Waits until you press any key on the keyboard.

key_pressed

Succeeds if any characters are waiting in the keyboard buffer otherwise fails.

key_accept(X1)
key_accept(X1,X2)

Unifies the ASCII code of the next character waiting in the keyboard buffer with **X1** and unifies its scancode with **X2.** The scancode refers to the position of the key pressed on the keyboard rather than to the character it triggered. If the buffer is empty "**key_accept**" prompts you to enter a character.

egalf(X1,X2)

Succeeds if **X1** and **X2** are the same objects otherwise fails. Two objects can be unifiable but not the same, e.g.

egalf(X,Y).	**fails**
egalf(X,X).	**succeeds**
egalf([1,2],[1,2]).	**fails (!)**
eq(X,Y),egalf(X,Y).	**succeeds**

make_ground
make_ground(X)
make_ground(X,I)

Unifies each unbound variable in the term **X** with a unique, newly generated constant called ground constant. A ground constant is a constant different from any other constants of the system. The ground constants are numbered from 0 (or from I if given) to 2^16-1. The ith ground constant appears in the output as Xi. If it is called without arguments it resets the ground constant counter to zero.

abort

Aborts the execution.

set_option(X1,X2)
set_option(X1,X2,X3)

Sets the option **X1** to value **X2** (see **OPTION** menu description in chapter "Programming Environment"). If the third argument is given the previous value of the option **X1** is unified with **X3**. The meaning of parameters is as follows:

X1 = 0 sound	**X2** = 0 off, **X2** = 1 on
X1 = 1 error on undefined	**X2** = 0 off, **X2** = 1 on
X1 = 2 tail recursion opt.	**X2** = 0 off, **X2** = 1 on
X1 = 3 acknowledge	**X2** = 0 off, **X2** = 1 on
X1 = 4 print	**X2** = 0 off, **X2** = 2 trace, **X2** = 3 dialog, **X2** = 4 all

garbage_collection

Performs explicitly called garbage collection. You will see a little red window appear on the screen signalling memory management.

random(X)

Generates a pseudo random number between [0,1] and unifies it with **X**.

system(C)

C must be an atom representing a system command in the current operating system. Performs the appropriate system command and return to the CS-PROLOG run immediately. If the execution of the operating system command was successful it succeeds otherwise fails. You have to reserve some memory for the system commands when you start the CS-PROLOG system using the **"/dosmem=N"** option where **N** denotes the amount of memory in Kbytes to reserve for system command execution.

port_byte(I,IV)

I must be an integer within the range [0, 65535]. **IV** must be either an integer within the range [0, 255] or an unbound variable. This predicate serves for executing input/output through hardware ports. **I** represents the address of the hardware port. If **IV** is an integer then its numerical value as a byte is sent to the **I**th hardware port. If **IV** is an unbound variable then a byte is fetched from the **I**th hardware port and its numerical value is assigned to **IV**. Always succeeds.

5. Parallel Execution

5.1 Parallelism In CS-PROLOG

CS-PROLOG is an extended PROLOG system which allows parallel execution of PROLOG goals. A "process" is assigned to each simultaneously executed goal. A process is a PROLOG subprogram and works like an ordinary PROLOG program except the handling of parallel control predicates.

In a monoprocessor environment the execution of the processes is controlled by an internal scheduler (quasi parallel execution) using the next event simulation discipline.

The processes can be created and deleted dynamically during runtime with the built-in predicates **"new"** and **"delete_process"**. In order to realize the synchronization between processes they can send messages to and receive messages from other processes by using the built-in predicates **"send"** and **"wait_for"**. No communication through common logical variables or by modification of the common database is supported.

Another extension in CS-PROLOG is the explicit notion of time which can be used for discrete simulation. Time is implemented as a counter clock. A CS-PROLOG time unit does not correspond to real time nor to cpu time. In a monoprocessor environment time is global counter clock called systemtime for all processes in the system.

Processes can be deactivated for a definite or indefinite period of time using the **"hold"** predicate. A process deactivated by a **"hold"** predicate is said to be in **"waiting"** state. A waiting process will be reactivated when the given period of time has elapsed.

Process is said to be in **"blocked"** state when it called a **"wait_for"** predicate and waits for the arrival of a message.

A process can wait for a message or for activation. This latter means that the process is to be activated/reactivated at a given time moment (in simulated time). The process is activated when the internal (simulated) clock reaches the waited time.

In the monoprocessor version of CS-PROLOG system the internal scheduler controls the execution of processes because only one process can be "active" at a time. If there are more processes in the system they are surely in waiting state. Scheduler allows the active process run until the process reaches any of predicates causing its deactivation or it solves its task. Then another activable process is chosen. If there isn't activable process in a given moment systemtime is

incremented to a point where a waiting process has to be activated.

Since the forward execution of processes results a sequentialized order of execution backtrack is executed backward in this path. This means that everything is undone until the last choice point even through process change or message sending/waiting operations.

5.2 The Scheduler of CS-PROLOG

(1) Active process

The process under execution is called the **"active process"**. On a single-processor computer there may only be one **"active process"** at a time.

(2) Waiting process

Processes that have not been activated or are to be reactivated at a given time (because they are suspended for a certain time interval) are called **"waiting processes"**.

Every **"waiting process"** has a time **"t"** attached which shows when the next activity of the process is to be executed. On the waiting list the processes are ordered by this time **"t"** in increasing order. The time **"t"** is called the activation/reactivation time of the process.

(3) Blocked process

Processes that are waiting for a message are called **"blocked processes"**.

(4) Activable processes

At a given point of time **"t"** the following processes are activable:

(a) all the processes of the waiting list, the activating/reactivating time of which is t;

the processes of the blocked list, that have received an appropriate message;

(b) if there are no more processes of type (a) and the activating/reactivating time of the first member of the waiting list is **"t1"** (**t1 > t**) then every process in the waiting list that is entered with time **"t1"** as activating/reactivating time

In the latter case the system time is moved to time point **"t1"**.

Now the algorithm of the scheduler of CS-PROLOG will be the following.

Originally system time is 0.

(i) The active process will be the first one that can be activated.

(ii) Every activity of the active process is executed in sequence until an activity that causes suspension is met or all the activities of the process have been executed (the process terminates successfully).

(iii) If the case is the latter one, that is there are no more activities in the active process and the waiting list is not empty (after trying to find activable processes), then we go to (i).

(iv) If there are no more activities of the active process and all the waiting, message, blocked lists are empty the CS-PROLOG program has **successfully terminated**.

(v) If the active process has terminated successfully but the blocked list is not empty and there is no process that can be activated then **backtracking**.

(vi) If the active process is suspended (by a **"hold"**, **"wait_for"**, **"wait_for_dnd"** predicate) and after trying to find activable processes the waiting list is not empty then (i) is executed again.

(vii) If the active process is suspended (by a **"hold"**, **"wait_for"**, **"wait_for_dnd"** predicate) and the waiting list is empty (no activable process) but the blocked list is not empty then there is no process that can be activated so backtracking begins.

System time is always the activation time of the active process.

5.3 Process Manipulating Predicates

new(GOAL)
new(GOAL,NAME)
new(GOAL,NAME,START)
new(GOAL,NAME,START,END)

Creates a new process with goal **GOAL**, name **NAME**, starting time **START**, and termination time limit **END**. If the **NAME** argument is missing the system assigns an internal name to the new process. If there are unbound variables in the argument **NAME** these are instantiated with different ground constants (see **"make_ground"**). If **START** is missing or is a variable it is set to the actual systemtime (that is CS-PROLOG time, not cpu time or real time!). If **END** is missing or is a variable it is set to 10^50.

send(MESS,PROC_LIST)

Sends a message **MESS** to the processes whose names are on the **PROC_LIST**. If the message is sent to one process only the PROC_LIST must be a one element list. If you want to send a message to all existing processes it can be done using an unbound variable as PROC_LIST.

wait_for(MESS)

If a process executes a **"wait_for"** predicate and there was a message **M** sent to it using the **"send"** predicate (see above) and **MESS** and **M** are unifiable then the unification takes place and the waiting process continues its execution. If there was no such message the process is suspended until an appropriate **"send"** is executed by another process.

wait_for_dnd(MESS)

Does the same thing as **"wait_for"** except that if the first message received leads to failure and backtracking, taking in account the other messages that might have arrived also fails the process is suspended waiting for further messages. Thus the suspended process continues backtracking only after a global dead-lock. (This predicate is non-deterministic and backtrackable.)

hold(T)

The execution of the active process is suspended for **T** time units. The system will activate another process. If all processes are suspended - e.g. some via **"hold"**, others via **"wait_for"** - time will advance to the next point where another process can be activated. Then this process (or one of them if there are several) will be activated.

delete_process(P)

All processes whose names can be unified with **P** are deleted.

active_process(AP)

Unifies **AP** with the name of the active process.

systemtime(ST)

Unifies **ST** with the actual CS-PROLOG simulated time.

message_arrived(X)

If there is a message sent to the active process and unifiable with **X** then succeeds otherwise fails. **X** is not unified with the message even if the predicate succeeds!

termination_time(P,T)

Unifies **T** with the prescribed termination time limit of process **P**.

reactivation_time(P,T)

Unifies **T** with the prescribed reactivation time of the process **P** if the process **P** has been suspended by **"hold"** otherwise fails.

run(GOAL)

Initializes a CS-PROLOG execution that uses parallel features. A program that uses any of the above listed predicates for time dependent execution must be executed with the "**run**" predicate.

6. The Programming environment

6.1 Projects

In CS-PROLOG there is no real modularity but the environment supports development of programs consisting of several (more than one) files. The set of files forming a program we call **project**. The file containing the file names of a project we call 'project file', the PROLOG files are called 'source files'.

There are some restrictions when working with projects. One partition (clauses with the same name) must not be split into different source files. If you use operator declarations the order of source files (in the project file) can be important. If an operator is declared in a source file you can use it in the rest of sources following this one.

All source file names must be different names.

6.2 Focus

In the environment of CS-PROLOG there is always a selected source file and a particular selected clause inside this file (unless the project is empty or a source file is empty). Several actions are working with this selected file or clause (such as 'exclude(file)' or 'delete(clause)'). This selection we will call **focus**ing, the selected clause is the clause we are **focus**ed on.

The clause we are focused on is highlighted in the observer window under the main menu. You can move the highlight (the focus) inside the current source file using the following keys:

Up, Down	Moves highlight one clause up or down.
PgUp, PgDn	Moves highlight one page up or down.
Home, End	Moves highlight to the first or last clause in the actual window.
Ctrl-Home, Ctrl-End	Moves highlight to the very first or to the very last clause of the file.

There can be comments in a source file. A comment belongs to the clause following it. So you can enter, modify and

delete comments only focusing on this clause and then modifying it.

6.3 Menus

In the environment you can select the action you want to perform using menus. A menu is a list of identifiers, one of these names is highlighted. You can select the desired item either by positioning the highlight with **Right** and **Left** in horizontal menus, with **Down** and **Up** keys in vertical menus, and then pressing **Enter**. Pressing the upper-case letter contained in the given item will select this item as well.

In several cases there are some menu items which are not selectable (e.g. delete(clause) if there is no clause in the source file). These items are displayed in different color.

You can always return from a menu to the previous level of the environment pressing the **Esc** key. This is not true for the main menu. You can leave the CS-PROLOG environment only selecting the **Quit** item. In a vertical menu you can escape from it not only with **Esc** but also pressing the **Left** or **Right** key. In this case the next vertical menu is pulled down (if any).

6.4 Browsers

If you want to choose a file name or a predicate name (e.g. to load it, or to focus on it) using a browser you have the possibility to select it from the list of all existing names. This means that you don't have to type in the name (of a file or of a predicate). You get all possible names in a window, the names are sorted in alphabetical order. One name is highlighted, you can use cursor keys to choose from the list. If all names cannot be displayed in one window, a '<' or '>' sign indicates in the window corner that there are more items in that direction. The function keys for the browsers are the following:

Enter:	Select the highlighted item.
Esc:	Quit the browser without selection.
Cursor keys:	Move highlight.
PgUp, PgDn:	Display the previous or next portion of items.
Home, End:	Move highlight to the first or last item in the window.

Ctrl-Home, Ctrl-End: Move highlight to the very first or very last item in the browser.

A,B,C, ... ,Z: Move highlight to the next name beginning with this letter.

There are three special browser types in the system, the file-browser, the focus-browser and the breakpoint-browser. The latter one will be described in detail in the chapter dealing with the debug facility.

6.4.1 File browser

File browsers are used to select a file name from the disk. When it is called there is a pattern given that defines which files are to be displayed. E.g. the pattern '*.pro' means all files with extension 'pro'.

The file browser is a double browser. Two windows appear on the screen. In the lower one the file names in the current working directory are displayed that match the specified pattern. In the upper window all subdirectories of the current working directory are listed including the '..' (parent) directory. You can select a file name in the lower window, using the function keys described in the previous section, but if you want to change directory press the

Tab

key. You will have the possibility to select a new directory name. After selecting a directory name the new list of files and subdirectories is displayed. To enter a path name explicitly press the **Tab** key again in the upper window.

6.4.2 Focus browser

When you want to select a clause in the program to focus on you can use the focus browser. If there is only one source file in the project a normal browser will be displayed. If there are more source files a double browser is used (similarly to the case of the file browser). In the lower window the clause names of the current source file are listed, in the upper one the source file names are displayed. You can get into the upper window pressing the

Tab

key. After selecting a new source file the new list of clause names is displayed in the lower window.

6.5 Editors

In the CS-PROLOG environment when you enter a text the built in editor is used. You can either type characters or execute an editing function pressing a function key listed below. There is one exceptional window - the clause editing window (used for entering and modifying clauses) - where some keys have a special meaning.

Esc: Quit the editor window without entering the text.

Enter: (normal windows) Quit the editor window accepting the entered text.

Enter: (clause edit window) Move cursor to the beginning of the next row.

F10, Ctrl-X, Alt-X, Ctrl-Enter: (clause editing window) Quit the editor window accepting the entered text.

Cursor keys: Move the cursor in the window.

Home, End: Move the cursor to the top-left or bottom-right corner of the window.

Ctrl-Left, Ctrl-Right: Move the cursor to the beginning or end of the current line.

Ins: Switch between insert or overwrite mode.

Del: Delete the character in the cursor position.

Backspace: Delete the character preceding the cursor position.

Tab, Backtab: Move cursor to the next or previous tabulator position.

F1: Insert an empty line under the current line.

F2: Delete the current line.

F3: Split the current line at the position of the cursor.

F4: Join the current line with the next one.

6.5.1 The scrap buffer

The scrap buffer contains a character string. This string can be created with **Copy to scrap** or **Cut to scrap** command and it can be modified with the **Edit scrap** command (for the description of these commands see the **Edit** chapter). The scrap buffer can be inserted in an editor window using the **F5** or **F6** keys:

F5, F6: These functions are available only when using the **Edit-Enter** or **Edit-Modify** menu-items. You can insert the scrap-buffer. **F5** simply inserts the scrap without modifying the rest of the editor window. **F6** scrolls down the window under the current line to make room for the scrap.

The scrap buffer is very useful when entering clauses that are very similar to each other.

6.6 The helpkey

On any place Pressing the helpkey you can get information about the environment anywhere. This key is **Alt-h** by default but any other key can be chosen for this purpose in the Setup submenu. So you can read a detailed description about the function you are currently using.

The help texts are stored in a file named:

csprolog.hlp

This file is searched in the current working directory and in the directory set in the environment variable CSPHOME. If this file is not found the help facility is not available.

6.7 The main menu

In this menu you can choose one of the following activities:

File: Changing the structure of the project.

- Load a new project from the disk.
- Load a new source file to the project.
- Add a new, empty source file to the project.
- Save the project.
- Save the source file.
- Delete a file from the project
- Select the source file to focus on.
- Rename the source file.
- Clear the data base.

Edit: Changing the current source file.

- Enter a new clause.
- Modify or delete a clause.
- Load clauses from a file on the disk.
- Edit the source file with an external editor.
- Modify the scrap buffer.
- Focus on a specific clause.
- Search a clause containing a specific string.

eXec: Execute a goal-sequence.

- Run a goal.
- Debug a goal.
- Run a goal and display the variable instantiations.

Option: Set, load or save the values of the CS-PROLOG options.

Setup: Change some global parameters of the environment.

Quit: Exit to the operating system. If you have modified files that have not been saved you will be given a chance to cancel the **Quit** command and save your files.

Help: Get more help.

6.8 File submenu

In this submenu you can change the structure of the project.

6.8.1 Load project

This action loads source files described in a project file to the environment. The current project is deleted (if any). If the current project has modified and not saved source files the user is asked for confirmation of the deletion.

The project file is a simple text file with file names of the source files. If the extension of a source file is the same as the 'source file pattern' (see setup submenu), the extension can be omitted. (This extension is by default: 'pro'.) When saving a project the system creates such a project file.

Selecting the **load project** submenu item you are asked for the project file name. If the extension of the file is the same as the 'project file pattern' (see setup submenu) the extension can be omitted. (This extension is by default: 'prj'.) You can enter a partially defined file name using the wildcard characters '*' and '?'. In this case the file browser will be invoked with this name as the pattern to match (see the description of the file browsers). Entering an empty line will invoke the browser with the 'project file pattern'. So if you leave the default value of this pattern entering an empty line you will get in the browser all files with the extension 'prj'.

If the system encounters a syntactically wrong clause an error message is displayed and you can correct the clause in an editor window. **Esc**aping from this editor you can ignore the incorrect clause.

After loading a project the focus is set to the first clause of the first source file.

6.8.2 Load file

Loading a source file means adding a new source file to the project. If in the new file there is a clause with the same name as a clause in an old file this new clause is not added. After the load the focus will be on the first clause of the new source file.

Selecting the **load file** submenu item you are asked for the source file name. If the extension of the file is the same as the 'source file pattern' (see setup submenu) the extension can be omitted. (This extension is by default: 'pro'.) You can enter a partially defined file name using the wildcard characters '*' and '?'. In this case the file browser will be invoked with this name as the pattern to match (see the description of the file browsers). Entering an empty line will invoke the browser with the 'source file pattern'. So if you leave the default value of this pattern entering an empty line

you will get in the browser all files with the extension 'pro'.

If the system encounters a syntactically wrong clause an error message is displayed and you can correct the clause in an editor window. **Esc**aping from this editor you can ignore the incorrect clause.

6.8.3 New file

Selecting the **new file** submenu item you can create a new - empty - source file in the project. The system asks for the name of the new file. If the extension of the file is the same as the 'source file pattern' (see setup submenu) this extension can be omitted.

6.8.4 Save project

When you want to save a project you have to enter the output file name. If the project was created with **load project** the name of the loaded project file is displayed so you simply have to send it with the **Enter** key (of course only if you don't want to change the file name).

When saving a project only those source files are saved which were modified.

6.8.5 Save file

You can save a source file alone without saving the project. You are asked for the output file name in the window you will find the original file name. If you change this name that does not change the source file name in the project.

6.8.6 Exclude file

Selecting this menu item means the deletion of the current source file from the project. It **does not** delete any file from the disk! The clauses of the source file are deleted only from the data base of the CS-PROLOG environment.

6.8.7 Next file

This action serves for focusing on the next source file. If you want to focus on a named source file use the **Select file** command described in the next chapter.

6.8.8 Select file

Choosing the **Select file** menu item you get a browser where you can select the source file to focus on.

6.8.9 Rename file

You can change the source file name using this command. The renaming is done only in the CS-PROLOG environment. There is no change on the disk! When the project or the file is saved the new file name is used.

6.8.10 New system

This command is used to initialize the whole CS-PROLOG environment. The project, source files, all dynamically added data will be deleted. The state of the environment will be the same as it was when you started the CS-PROLOG.

6.9 Edit submenu

The commands in this submenu serve for modification of the current source file.

6.9.1 Enter

When you choose this menu item an editor window is opened where you can type in a new clause. If the clause syntactically is not correct an error message is displayed and you have to correct the clause. Only one clause can be entered at a time.

The new clause is inserted following the focused one. There is one exception: when a partition with the same name as the new clause exists already and the focused clause does not belong to this partition. In this case the new clause is

inserted as the first clause of this partition if the focus is above the partition and it is inserted as the last clause if the focus is under the partition.

After entering a new clause the focus is always set to itself.

6.9.2 Modify

You can modify the focused clause. It is not allowed to change the name of the clause. (To copy a clause (with changes) use the **Copy scrap** and then **Enter** and then the **F5** or **F6** key.)

6.9.3 Delete

Selecting this menu item the system deletes the focused clause (without warning). The clause following the deleted one will be the new focused clause (if the last clause is deleted the focus will be put on the previous one).

6.9.4 Insert

Selecting the **Insert** command you can enter new clauses in the editor window. Unlike the **Enter** command this command allows to enter more than one clause. Every clause has to begin in a new line.

The new clauses are appended to the end of the source file even if the focus is not on the last clause. The first of inserted clauses will be the new focused one.

6.9.5 Load text

This command enters new clauses from a disk file. You can specify the file to load in the same way as it is described in the **Load file** chapter.

The difference between the **Load text** and **Load file** commands is the following. **Load text** loads clauses into the current source file while **Load file** adds the file as a new source file to the project.

The new clauses are appended to the end of the source file even if the focus is not on the last clause. The first of loaded clauses will be the new focused one.

6.9.6 Edit external

If you want to make major changes in a source file a text editor may be more convenient than the editing facilities of the CS-PROLOG environment. This command makes it possible. (It contains an implicit **Save file** before editing, and **Load file** after editing.)

You have to specify the text editor you want to use in the **Setup - File editor** menu item.

6.9.7 Copy to scrap

The character string of the focused clause is copied to the scrap buffer. (For the definition of the scrap see the chapter describing the Editor facilities).

6.9.8 Cut to scrap

The character string of the focused clause is copied to the scrap buffer and then this clause is deleted. This command has an identical effect as the **Copy to scrap**, **Delete** command sequence.

6.9.9 Edit scrap

You can modify the scrap buffer in an editor window. There are no syntactical restrictions for the content of the scrap.

6.9.10 Focus

You can select the clause to focus on using the cursor keys in the main menu. If you want to specify explicitly the name of the clause to focus on then use this command. Entering an empty line as name invokes the focus browser (see the focus browser chapter) where you can choose from the list of the clause names.

6.9.11 Search

Use this command to find a clause containing a specific string. The search begins in the clause following the focused

one (so to search in the entire source file, you have to focus on the top clause first). The search is performed only in the current source file. To search in another source file you have to focus on it first.

If an empty search string is entered the system displays the previous search string (if any) and then it can be reentered or modified.

6.10 Exec submenu

This submenu serves for executing CS-PROLOG goal-sequences. When you select one of the **Run, Debug** and **Solution** menu items you can enter a goal-sequence to execute. This sequence can be max. one line long. If you send an empty line the system displays the previous goal-sequence which can be reentered (after a modification if wanted).

6.10.1 Run

This command simply executes the goal-sequence. The "system dialog window" is opened by default. After the execution the environment indicates its success or failure displaying a **SUCCEED** or **FAIL** message on the screen.

6.10.2 Debug

This menu item serves to trace CS-PROLOG goal-sequences with the interactive debugger. The usage of the debugger is in the next chapter.

6.10.3 Solution

If you want to execute a CS-PROLOG goal-sequence to get the value of some output variables then use the **Solution** menu item. It executes the goal-sequence and after the successful termination the matched values are displayed for all variables in the goal-sequence. Then then system asks:

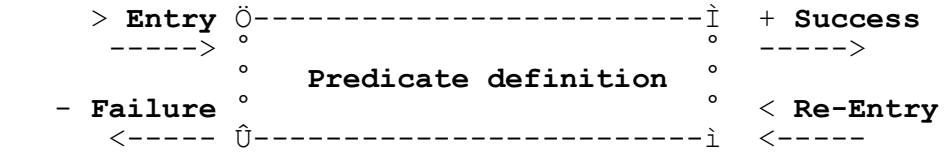
```
Continue(y,n)
```

Answering with 'y' key will cause a backtrack and another solution will be displayed (if any).

6.11 Debugging CS-PROLOG programs

6.11.1 The "box" model

To explain the CS-PROLOG interactive trace facility it is helpful to define the following "box model" of a predicate. Each predicate is enclosed in a box. The box has two entering ports and two exiting ports.



The symbols **">"**, **"+"**, **"<"**, and **"-"** are the symbols of the four ports. The **Entry** port is used when a predicate is evaluated at the initial invocation of the predicate. The **Success** exit is used after successful execution. The **Failure** exit is used after failed execution. The **Re-Entry** port is used during back tracking when CS-PROLOG tries to find new alternatives. If a predicate is traced all ports are displayed on the trace screen.

6.11.2 The interactive trace

You have to enter your goalsequence in the **Debug** command. CS-PROLOG will stop at the entry point of the first predicate to be executed. Now the user can control further evaluation with the following keys:

Down or **Enter** Stop at entry port of the next predicate to be evaluated. If the current predicate has a body the first call in that body is next to be traced.

Right Stop at entry port of the next predicate called after the termination of the current one. In contrast to **Down, Right** does not trace the body of a predicate but only the predicates after its own evaluation.

Up Stop at the next entry port encountered after terminating the execution of the parent of the traced predicate. This means no further stop while executing the body of the parent.

G Go without stop to the next break point (see below how to set break points).

Grey + (The plus key on the numeric keypad!) Force the executed call to succeed. The variables are not instantiated.

Grey - (The minus key on the numeric keypad!) Force the executed call to fail.

A, Q or Esc Abort the execution.

F5 Switch between trace screen and the main screen of the environment. You get into the main menu, where you can change the focus and edit your program. To return to the trace, select any item except **Edit**, or press **Esc**.

F6 Switch between current screen and the output screen. Press any key to return to the trace.

S Set trace and break points (see below).

6.11.3 Setting breakpoints

The predicates of the program (both the built-in and PROLOG predicates) can be marked as a **breakpoint tracepoint** or **gopoint**. If a **breakpoint** predicate is called the trace stops before calling it and the user gets the control. The trace does not stop on **tracepoints** but all ports (see box model) are displayed. **Gopoint** is similar to the **breakpoint** but when it is called first time it looses this marking.

After typing **the S** command (at the trace level), the user is asked whether he wants the list of built-in system predicates or one of user defined predicates. Answer with **b** or **p**. Next a new browser-window will appear with a list of the specified predicates in alphabetic order. The first name is highlighted. You can move the highlight with the cursor keys and **PgUp** and **PgDn** keys, similarly as in other browsers. Several letter-keys have here a very special function (so pressing a letter-key does not mean highlighting the next name beginning with it).

F Search a name. You are asked for a string, and the highlight will be put on the name which has the longest beginning common slice with this string.

T Set the highlighted predicate to be a **tracepoint**.

B Set the highlighted predicate to be a **breakpoint**.

G	Set the highlighted predicate to be a **gopoint.**
U	Unmark the highlighted predicate.
C	Unmark all predicates
Enter, **Esc**	Finish breakpoint setting. Both keys have the same effect, **Esc** does not undo the markings.
1,2 ... 9:	You can get the browser with as much columns as the key you pressed. (The **"~"** sign at the end of a name means that this name is longer then the length oh the field.)

6.11.4 Debugging Parallel Execution

If there is more than one process in a CS-PROLOG program the trace screen can be split into several windows. The trace of different processes is displayed in different windows. The user can specify the number of trace windows in the **Setup** menu. This number can be set to **1, 2, 4, 6, 8, 9** or **12.** When a process is created the system asks whether this process is to be traced or not. The answer must be **y** or **n.** If **y** is typed the next free trace window is used to trace this process. The name of the process is printed at the top of the window.

6.12 Option submenu

There are five CS-PROLOG system state variables, called options, that have effect on the execution of goal. In the **Option** submenu you can change the settings of these variables, save the current settings to a disk-file, or load settings from a previously saved file.

The options, and their possible values are the following (the defaults are underlined):

Sound	**Off**	No sound generated
	On	Sound generated after several functions.
Error on undefined	**Off**	If a predicate is called and there is no definition for it then the predicate fails.
	On	In this case an 'undefined_predicate' error is signalled.

Tail recursion opt.	**Off**	No tail recursion optimization is performed.
	On	Tail recursion optimization is performed during the execution.
Printer output	**Off**	No default output on the printer.
	Trace	Trace outputs are printed.
	Dialog	All input/output in dialog windows are printed.
	All	All input/output are printed.
Acknowledge	**On**	Hold output if a window screen is full. The execution continues when a key is pressed.
	Off	Continuous output.

6.13 Setup submenu

In the **Setup** submenu you can change the values of some state variables of the CS-PROLOG environment. So these settings have no effect on the execution only on the environment.

6.13.1 External editor

You have the possibility to modify a source file with your favorite text editor without leaving the CS-PROLOG environment (see the **Edit external** item in the **Edit** submenu). You have to specify here the editor you want to use. It can be any executable file name. When you select the **Edit external** function, the environment will issue the following operating system call:

Editor filename

where **Editor** is the name you entered in the **'Setup-External editor'** function, and **filename** is the source file name.

6.13.2 Source pattern

The source pattern is used in source file-browsers to select the files to display (in **Load file** and **Load text** functions). The pattern can contain the wildcard characters '*' and '?'. These characters have the same meaning as in the operating system. E.g. the pattern

hu*.??

means all file names that begin with 'hu' and have two character extension.

If the source pattern has the form:

*.EXT

where 'EXT' is an extension not containing wildcard characters, this extension is used as default extension in all places where source files are specified. So in this case you do not have to give the extension if it is the same as the default. The default extension is used in project files, in **Load file**, **New file**, **Save file**, **Rename file**, and **Load text** functions.

The default value for the source pattern is

*.pro

so the default 'default extension' is 'pro'.

6.13.3 Project pattern

Project pattern is the same for the project files as the source pattern is for the source files. The project pattern and the default project extension (if any) is used in **Load project** and **Save project** functions. The default value for the project pattern is

*.prj

6.13.4 Edit window size

The clause editor window is used when you are entering or modifying a clause or the scrap buffer. (See the **Edit** submenu, **Enter**, **Modify**, **Edit scrap** items.) You can set the row size of the clause editor window using this **Setup** menu item. Enter simply a number between 3 and 18. The default value is 8.

6.13.5 Trace windows

The trace of different processes can be directed into different windows (see the description of the interactive trace). You can specify here the number of windows used by the trace (max. 12). The default value is 1.

6.13.6 Deadlock detection

This command has effect only when you are debugging a parallel program. If the **deadlock detection** is set to **On** then when a deadlock situation occurs, while you are debugging a parallel program, a special window opens and informs you about the current state of the scheduler's internal lists. Pressing a key you can continue the debugging. (Backtrack begins from the deadlocked point.) By default the **deadlock detection** is set to **Off** so no information appears in a deadlock situation.

6.13.7 Colors

You can set the colors of the windows used by the environment. The following window groups can be set separately:

Menus	The menu and browser windows. You have to set the color of the highlight as well.
Editors	The editor windows, all windows where a text is entered.
Observer	The window under the main menu where the clauses are displayed. You have to specify the color of the highlight as well.
Error	The window for error messages.
System dialog	The window which is opened by default before executing a goal-sequence. (This window has the PROLOG name "system dialog window").

When you have selected one of the window types above a sample window appears in the middle of the screen. You can change the background color pressing the **Up** or **Down** keys and the foreground color pressing the **Right** or **Left** keys. Press **Enter** to set the shown colors. When setting the colors of the menu or observer windows you have to select a color for the highlight as well.

After setting all colors you wanted to change return to the main menu pressing **Esc**.

6.13.8 Helpkey

You can change the default setting of the helpkey (**Alt-h**) to any other key. Just press the new help key. The system will ask for the confirmation of the change. Several keys have special meaning for the system (e.g. **F1-F6**, cursor keys, **Enter**, etc.), it is not advised to overload these keys, e.g. to use the **F1** key as the help key.

6.13.9 Save setup

If you have changed several setup parameters and you want to make these changes permanent you have to save the actual setup using this menu item. The system creates a file named:

csprolog.ini

containing the setup parameters. When the CS-PROLOG environment is called it looks for a file 'csprolog.ini' in the current working directory. If this file is found the setup parameters are set to values stored in the file. You can reset the default parameters deleting the file from the disk. Obviously in different directories there can be different 'csprolog.ini' files.

6.14 Help submenu

In this submenu you find three items. Choosing the **Help** item you get some basic information. With **Content** item you can list the titles of the help texts available. In the **Manual** item you have to enter a help text number and the system displays the help texts from the chosen one.

On the top of every help screen you see a headline containing the title, the total number of help screens and the serial number of the actual one. You can switch between help screens using the **Enter** key or **PgDn** key to get the next, **PgUp** key to get the previous and **ESC** key to finish. Pressing **Enter** on the last page finishes the help utility too.

7. Examples

This section contains some programming examples that demonstrate the use of time and parallel processes in CS-PROLOG as well as window handling and graphics. The code for each example is completely reproduced.

7.1 Bank Robbery

This example simulates a bank robbery where the two thieves, Jim and Dick, try to break into the MultiMoney Bank. Jim actually enters the bank while Dick is waiting outside delivering tools to Jim for opening a safe. The whole task is explicitly time limited. Both Jim and Dick are represented as (parallel) processes. Messages are being sent between these processes the first to report the kind of safe found inside the bank the second to deliver tools for opening that safe.

The following is Jim's task list. It tells him that in order to succeed in the bank robbery he has to climb into the bank, choose a safe, report the kind of safe to Dick who is waiting outside, wait for special tools for opening that reported safe, open the safe and finally take all valuables from inside the safe outside the bank. Note that "safe" is a predicate that is true if its only argument is a valid safe while "SAFE" is a variable with the name of an actual safe as its value.

```
jim_gets_the_money(BANK):-
   jim_climbs_into(BANK), chooses(SAFE),
   send(safe(SAFE), [dick]), wait_for(tools(TOOLS)),
   opens(SAFE,TOOLS),    outputs(SAFE,BANK).
```

This is Dick's task list:

```
dick_gets_the_money(BANK):-
   wait_for(safe(SAFE)), has(TOOLS,SAFE),
   send(tools(TOOLS),[jim]).
```

Following are three choices of a safe inside MultiMoney Bank:

```
chooses(wertheim).

chooses(milner).

chooses(chatwood).
```

There are tools for opening a safe available only for the Milner and Chatwood safe:

```
has(tool_set_a,milner).

has(tool_set_b,chatwood).
```

Climbing into the bank takes Jim five logical time units:

```
jim_climbs_into(BANK):-
   during(5).
```

Opening the Milner safe takes 40 logical time units, opening the Chatwood safe takes only 10 time units:

```
opens(milner,TOOLS):-
   during(40).

opens(chatwood,TOOLS):-
   during(10).
```

This is to report the result on the display:

```
outputs(SAFE,BANK):-
   systemtime(T),
   write("The thieves got the money from the safe"),
   write(SAFE), write(" in the "), write(BANK),
   write(" Bank at time "), write(T), write(.), nl.
```

"**during**" is a synonym for "**hold**":

```
during(T):-
   hold(T).
```

The toplevel predicate comes next. Two processes are initialized. "**dick**" and "**jim**" are the names of the processes. Their start time is 0 and their prescribed termination time is 25. Start the program with "**run(problem).**".

```
problem:-
   new(dick_gets_the_money(multi_money), dick, 0, 25),
   new(jim_gets_the_money(multi_money), jim, 0, 25).
```

Backtracking will occur because

there are no tools for the Wertheim safe

opening the Milner safe would take too long (40 time units for opening the safe plus 5 time units for entering the bank would exceed the time limit of 25 time units).

It is recommended to have at least two trace windows configured and trace the Dick and Jim processes.

7.2 Prime Number Generation

This is a very tricky implementation of the sieve of Eratosthenes algorithm. A global variable "**prime_number**" has either of the two values "**on**" or "**off**". If it is on the current number is a prime number. Every prime number will initiate a new process that will set the "**prime_number**"

variable to **"off"** at every multiple of itself starting from its square product (below p2 smaller multiples of that prime number are turned **"off"** by smaller prime numbers). Numbers are actually time units. Thus one could summarize that program as follows: time proceeds and every number not being a prime number will be turned **"off"** by a prime number that is a factor of that number. Prime numbers actually fall through that sieve of prime number processes. The program terminates when the stack overflows.

Next is a recoursive loop to check whether the actual number that is the actual CS-PROLOG time is a prime number. **"hold(1)"** causes to try the next number.

```
number_generation:-
   hold(1), is_prime_number, number_generation.
```

A number is a prime number if the **"prime_number"** variable is set to **"on"**. If this is the case print the prime number and initiate a new prime number process:

```
is_prime_number:-
   get_value(prime_number,off), !,
   set_value(prime_number,on).

is_prime_number:-
   systemtime(T), write(T), nl, TT is T * T,
   new(sieve(T),[sieve, T],TT).
```

Next is the predicate for a prime number process. It is activated only every T time unit which corresponds to all integer multiples of T.

```
sieve(T):-
   set_value(prime_number,off), hold(T), sieve(T).
```

Now follows the toplevel predicate. It creates the number generating process at time 1 with the name **"generator"**:

```
sieve_start:-
   set_value(prime_number, on), new(number_generation,
   generator, 1).
```

Start the program with **"run(sieve_start)."**.

7.3 Sorting In Time

This example program produces an ordered output of all numbers of the form 2^n * 3^m. The algorithm used involves four processes. The **"copy"** process to print out a number and recoursively wait for the next one; the processes **"mult2"** and **"mult3"** that produce the multiples of two and three; and the process "merge" that compares the multiples that it receives from **"mult2"** and **"mult3"**. This is surely a very inefficient way to produce the desired list but it is an excellent example

of how communication between processes works in the course of time.

The main predicate for the "**copy**" process:

```
copy:-
   wait_for(numb(X)), write(X), nl,
   send(numb(X),[mult2]), send(numb(X),[mult3]), copy.
```

Next comes the main predicate for the "**mult2**" and "**mult3**" processes. FAC is a parameter that makes "**pmult**" a generic predicate, i.e. one that stands for a class of predicates in this instance the class of multipliers.

```
pmult(FAC):-
   wait_for(numb(X)), Y is X * FAC, send(numb(Y, FAC),
   [merge]), pmult(FAC).
```

The main predicate for the "**merge**" process:

```
merge(N2,N3):-
   receive(2,N2,NN2), receive(3,N3,NN3),
   submerge(NN2,NN3).
```

The predicate to wait for and receive input from the "**mult**" processes:

```
receive(FAC,0,N):-
   !, wait_for(numb(N,FAC)).

receive(FAC,N,N).
```

"**submerge**" compares two numbers and branches:

```
submerge(N2,N3):-
   N2 < N3, send(numb(N2),[copy]), merge(0,N3).

submerge(N2,N3):-
   N3 < N2, send(numb(N3),[copy]), merge(N2,0).

submerge(N2,N3):-
   N3 = N2, send(numb(N3),[copy]), merge(0,0).
```

The toplevel predicate initiates the four processes and sends the first message:

```
goal:-
   new(copy,copy), new(pmult(2),mult2),
   new(pmult(3),mult3), new(merge(0,0),merge),
   send(numb(1),[copy]).
```

Start the program with "**run(goal).**".

7.4 Shortest Path Search

This example program finds the shortest path from a start node to a goal node via breadth-first search. The very interesting feature of this implementation is that the results are almost a side effect of the travelling of certain **"messenger"** processes along that network. The program has two parts. The first part the network is established from facts about connections between nodes. In the second part **"messenger"** processes - actually they are named **"mess(NODE, X)"** - are made to travel along the edges of the network. Every edge that has been travelled along is removed immediately. The time required to get from one node to the next is proportional to the distance encoded in the network. The messengers actually wait as much (logical) time in the program until they arrive at the destination goal. Every messenger that arrives at one node also deletes any other messengers that are in the same node.

The behavior of all the messenger processes can be summarized as follows: starting from a start node a messenger takes off in every direction. It is made sure that no path is travelled twice. The messengers one after the other arrive at their destination nodes. When they arrive they print out the node they are at and the time since they began their journey. From that node new messengers are sent out in any available direction who behave exactly as their parent. If there are no more nodes for any messenger to visit the program ends. The output is an ordered list of the distances from the start node to any other accessible node in the network.

The main predicate for the **"messenger"** processes:

```
messenger(NODE,LABEL):-
   hold(LABEL), delete_process(mess(NODE,X)),
   print_out(NODE), create_new_messengers(NODE).
```

The output predicate:

```
print_out(NODE):-
   systemtime(T), write_spaces(40), write("to node "),
   write(NODE), write(" is "), write(T), nl.
```

The predicate to initiate new messengers from a node in any available direction. It is recoursive:

```
create_new_messengers(NODE):-
   del_edge(NODE,NODE1,LABEL), !,
   new(messenger(NODE1,LABEL), mess(NODE1,X)),
   create_new_messengers(NODE).

create_new_messengers(NODE).
```

Next comes the predicate to remove paths that have been travelled along. It accounts for the fact that on a path from A to B you can also get from B to A. This saves 50% of the edges.

```
del_edge(NODE,NODE1,LABEL):-
   clause(edge(NODE,NODE1,LABEL),N),
   suppress_clause(edge,N), !.

del_edge(NODE,NODE1,LABEL):-
   clause(edge(NODE1,NODE,LABEL),N),
   suppress_clause(edge,N).
```

The start up predicate for the second part of the problem:

```
start_messengers(START_NODE):-
   write("The shortest path starting from node "),
   write(START_NODE), nl,
   create_new_messengers(START_NODE).
```

The predicate to construct the network from connection data:

```
construct_graph:-
   delete_partition(edge), connected(A,B,L),
   add_clause(edge(A,B,L)), fail.

construct_graph.
```

The toplevel predicate to start the program:

```
path_problem:-
   construct_graph, start_messengers(1).
```

Start the program with **"run(path_problem)."**.

Connection data. First two arguments are node numbers the third is a numerical measure of distance:

```
connected(1,2,25).

connected(1,3,10).

connected(1,7,100).

connected(2,5,10).

connected(2,4,40).

connected(3,4,15).

connected(4,7,35).

connected(5,6,35).

connected(5,7,25).
```

7.5 Effects Of The "wait_for" Predicates

This is a short example to illustrate the differences between **"wait_for"**, **"wait_for_nd"**, and **"wait_for_dnd"**. Process **b** checks whether it gets a message **m(3)**. If "wait" is **"wait_for"** **b** fails after trying the message **m(1)**. No backtracking before the message. If "wait" is **"wait_for_nd"** process **b** fails after trying **m(1)** and **m(2)**. Here is no backtracking before commitment on process **b**. If "wait" is **"wait_for_dnd"** process **b** fails after trying the messages **m(1)** and **m(2)**. Then process **e** gets selected, it activates process **c** which in turn sends **m(3)** to process **b**. Now process **b** succeeds. Start the example with a **"run(a)"** call.

```
a:-new(b,b), new(c,c), new(d,d), new(e,e).

b:-wait(m(X), write(tried(X)), nl, X=3, write(ok).

c:-wait_for(mm), send(m(3), [b]).

d:-send(m(1), [b]), send(m(2), [b]).

e:-send(mm,[c]).
```

7.6 Graphics Example

This program illustrates the use of some of the graphic predicates. Start the drawing of six cubes with **"a"**.

```
repeat(I).

repeat(I) :-
   I > 1, II is I - 1, repeat(II).

a :-
   graphic(on), init_indicator, clear_gr_screen, init,
   eagle_wake, draw, pause, graphic(off).

init :-
   background(0), palette(0).

eagle_wake :-
   wake(0), wake(1), wake(2), wake(3), wake(4),
   wake(5), view_point([0.0,700.0,2000.0]), pen(up),
   backward(60), backward(300,1), backward(300,4),
   forward(300,2), forward(300,5), turn(90),
   backward(300,3), backward(300,4), backward(300,5),
   turn(-90), tilt(30), forward(180), tilt(90),
   pen(down).

draw :-
   cube.
```

```
square :-
   repeat(4), forward(180), turn(90), fail.

square.

cube :-
   square, fill_square(2), forward(180), tilt(90),
   square, fill_square(2), forward(180), tilt(90),
   pen_color(3), square, fill_square(3), pen_color(1),
   square, forward(180), tilt(90), square,
   forward(180), tilt(90).

fill_square(C) :-
   pen(up), turn(10), forward(50), fill(C),
   backward(50), turn(-10), pen(down).
```

7.7 Windows And Menus

This program displays a menu on the screen. The menu contains all arguments for the predicate **"create_window"**. Using the cursor keys and the Enter and ESC keys it is possible to assign values for each of the arguments. Selecting the **"Draw"** item a window according to the chosen parameters is displayed. The appropriate call of **"create_window"** is shown in another window. Pay attention to the use of **"open_level"** and **"close_level"**. Start the whole thing with **"main"**.

```
main:-
   init_setting, setting("Draw").
```

"init_setting" initializes nine global variables that hold the information about the window to be created and creates itself four windows for the input/output. **"create_ver_windows"** creates another five windows for the five items of the main menu.

```
init_setting:-
   set_value(frame,0), set_value(back_ground,0),
   set_value(fore_ground,7), set_value(intensity,0),
   set_value(blinking,0), set_value("Top row",0),
   set_value("Left col",0), set_value("Row size",3),
   set_value("Col size",3),
   create_window(0,0,3,80,366,main_w),
   open_window(main_w), create_ver_windows,
   create_window(8,60,3,5,366,coord_w),
   create_window(8,30,3,20,1102,error_w),
   assign_text(error_w,1,5,"Wrong data"),
   create_window(24,0,1,80,110,show_set_w).

create_ver_windows:-
   create_window(2,3,7,18,366,"Frame"),
   create_window(2,13,10,12,366,"Background"),
   create_window(2,26,10,12,366,fOreground),
   create_window(2,41,4,18,366,"oTher attrs"),
   create_window(2,56,6,12,366,"Coordinates").
```

"setting" and **"continue"** make an alternating-recoursive loop for interaction with the main menu.

```
setting(FIRST):-
   hor_menu(main_w,1,[[3,"Frame"], [13,"Background"],
   [26,fOreground], [41,"oTher
   attrs"],[56,"Coordinates"], [71,"Draw"]],
   FIRST,14,X), submenu(X,I), show_setting,
   continue(X,I).

continue(0,I):- !.

continue(X,I):-
   is_num(I), !,
   nth_elem(["Frame","Background",fOreground,
   "oTher attrs","Coordinates","Draw"],X,N), N1 is N +
   I, setting(N1).

continue(X,I):-
   setting(X).
```

"submenu" and **"ver_data"** organize the pop-up menus of the main menu:

```
submenu(0,I):- !.           % ESC key pressed

submenu("Draw","Draw"):-
   get_attribute(ATTR), get_value("Top row",TL),
   get_value("Left col",LC), get_value("Row size",RS),
   get_value("Col size",CS),
   create_window(TL,LC,RS,CS,ATTR,show_w), open_level,
   open_window(show_w), pause, close_level.

submenu(X,I):-
   ver_data(X,ITEMS,FIRST), open_level, open_window(X),
   ver_menu(X,2,ITEMS,FIRST,14,I), set_data(X,ITEMS,I),
   close_level.

ver_data("Frame",["No
   frame","Single","Double","sIngle double",
   "dOuble single"],F):-
   get_value(frame,F1), F is F1 + 1 .

ver_data("Background",
   ["Black",bLue,"Green","Cyan","Red","Magenta", brOwn,
   "White"],F):-
   get_value(back_ground,F1), F is F1 + 1 .

ver_data(fOreground,
   ["Black",bLue,"Green","Cyan","Red","Magenta", brOwn,
   "White"],F):-
   get_value(fore_ground,F1), F is F1 + 1 .

ver_data("oTher attrs",[II,BI],1):-
   get_intensity(II), get_blinking(BI).

ver_data("Coordinates",["Top row","Left col",
   "Row size","Col size"],1).
```

The intensity menu item is variable:

```
get_intensity("no Intensive"):-
   get_value(intensity,0).

get_intensity("Intensive"):-
   get_value(intensity,1).

get_blinking("no Blinking"):-
   get_value(blinking,0).

get_blinking("Blinking"):-
   get_value(blinking,1).
```

Reading and storing user input:

```
set_data(X,Y,0):- !.   %ESC key pressed

set_data(X,Y,-1):- !.  %cursor-left pressed

set_data(X,Y,1):- !.   %cursor-right pressed

set_data("Coordinates",ITEMS,I):-
   open_window(coord_w), get_value(I,V),
   write_to_string(S,V), write_window(coord_w,1,1,S),
   edit_data(I).

set_data(X,ITEMS,I):-
   !, nth_elem(ITEMS,I,N), N1 is N - 1,
   set_data1(X,N1).

edit_data(I):-
   edit_data1(I), !.

edit_data(I):-
   open_level, open_window(error_w), pause,
   close_level, edit_data(I).

edit_data1(I):-
   read_window(coord_w,1,1,3,S,13), !,
   concat(S," ",S1), read_from_string(S1,V), is_int(V),
   set_value(I,V).

edit_data1(I).
```

```
set_data1("Frame",N):-
   !, set_value(frame,N).

set_data1("Background",N):-
   !, set_value(back_ground,N).

set_data1(fOreground,N):-
   !, set_value(fore_ground,N).

set_data1("oTher attrs",0):-
   !, get_value(intensity,X), X1 is 1 - X,
   set_value(intensity,X1).

set_data1("oTher attrs",1):-
   !, get_value(blinking,X), X1 is 1 - X,
   set_value(blinking,X1).
```

How to calculate the attribute number:

```
get_attribute(ATTR):-
   get_value(frame,F), get_value(blinking,B),
   get_value(back_ground,BC), get_value(intensity,I),
   get_value(fore_ground,FC), ATTR is 256*F + 128*B +
   16*BC + 8*I + FC.
```

This predicates shows the current values of the window parameters in the bottom line display.

```
show_setting:-
   get_attribute(ATTR), get_value("Top row",TL),
   get_value("Left col",LC), get_value("Row size",RS),
   get_value("Col size",CS), write_to_string(S,
   create_window(TL,LC,RS,CS,ATTR,window_name)),
   open_window(show_set_w),
   write_window(show_set_w,0,20,S).
```

"**nth_elem**" is an auxiliary predicate.

```
nth_elem([X|L],X,1):-
   !.

nth_elem([Y|L],X,N):-
   nth_elem(L,X,N1), N is N1 + 1 .
```

8. External C Interface

The CS-PROLOG system enables you to write your own built-in predicates in C language. A special C function set is supplied for parameter handling, memory allocation, choice point handling (for nondeterministic built_in predicates). The C interface function set is slightly different for the interpreter and the compiler in some cases. The following description will warn you when there is an incompatibility between them. The recommended compiler to be used is the Microsoft C compiler V 6.0. with the large memory model.

8.1 Global Items

The following global constant and structures are defined in the **"INTERFAC.INC"** header file.

Predefined constants used in interface functions

nil
true
false

Error numbers that can be returned by a built-in predicate

non_atom_argument
non_numeric_argument
memory_full
cannot_open_file
cannot_close_file
no_more_disk_space
syntax_error
wrong_arg_no
floating_point_error
non_opened_file
non_integer_argument
non_implemented
non_positive_argument
io_error
illegal_number
illegal_list

Flags representing different kinds of data in CS-PROLOG are:

t_empty
t_float
t_atom
t_fix
t_nil
t_struct
t_list

Basic data type for the internal data representation of CS-PROLOG is

csp_cell

For the compiler an additional data type is used

extb_choice_point

Two global cells representing the internal form of the empty list and unbound variable

```
csp_cell nil_cell;
csp_cell empty_cell;
```

The "**empty_cell**" can be used only to construct lists and structures containing unbound variables. Never use "**empty_cell**" as a parameter of "**unify_cell**". If you want to unify two different empty variables then build them into a structure or list and retrieve from this structure or list the cell representing the variables and then unify them.

8.2 The C Interface Function Set

```
int get_nth_arg(int arg_no, csp_cell *cell);
```

"**cell**" is the "**arg_no**"-th argument of the called built-in predicate. If "**arg_no**" is zero or greater then the actual number of arguments then "**wrong_arg_no**" is returned otherwise "**true**" is returned.

```
int put_nth_arg(int arg_no, csp_cell cell);
```

The "**arg_no**"-th argument of the called built-in predicate is unified with the "**cell**". It returns "**true**" is the unification was successful, "**false**" if it was not. Any other return number means error (memory full). If the unification is "**false**" then the variable bindings are not undone that means that if this function call fails then the built in predicate must return "**false**" as well.

```
int get_cell_type(csp_cell cell);
```

Returns the type of "**cell**":

t_empty	**unbound variable**
t_float	**float number**
t_atom	**atom (symbol)**
t_fix	**fix number**
t_nil	**empty list**
t_struct	**functional expression**
t_list	**list (non empty)**

int get_int_cell(csp_cell cell, int *value);

If the type of "**cell**" is not "**t_fix**" then returns "**false**" otherwise "**true**". The number represented by "**cell**" is assigned to "**value**".

int get_float_cell(csp_cell cell, double *value);

If the type of "**cell**" is not "**t_float**" then returns "**false**" otherwise "**true**". The float number represented by "**cell**" is assigned to "**value**".

int get_atom_cell(csp_cell cell, char **value);

If the type of "**cell**" is not "**t_atom**" then returns "**false**" otherwise "**true**". The string represented by "**cell**" is assigned to "**value**". It is very important that "**value**" is the pointer which points to the char sequence in CS-PROLOG memory tables. That means that the user must not change the content of this string! (You can only read this string.)

int get_list_head(csp_cell list, csp_cell *head);

If the type of "**cell**" is not "**t_list**" then returns "**false**" otherwise "**true**". The head of the list represented by "**cell**" is assigned to "**head**".

int get_list_tail(csp_cell list, csp_cell *tail);

If the type of "**cell**" is not "**t_list**" then returns "**false**" otherwise "**true**". The tail of the list represented by "**cell**" is assigned to "**tail**".

int get_file_cell(csp_cell f_cell, FILE **file_p);

If the type of "**f_cell**" is not "**t_atom**" then returns "**false**". Furthermore if "**f_cell**" does not represent a CS-PROLOG file identifier then returns "**false**". Otherwise it returns in its second argument a pointer to that **FILE** structure (defined in the "**stdio.h**" header file) which describes the named file.

```
int get_struct_functor(csp_cell s_cell,
                       csp_cell *name,
                       int *arity);
```

If the type of "**cell**" is not "t_struct" then returns "**false**" otherwise "**true**". The name and arity (number of arguments) of the functional expression represented by "**cell**" is assigned to "name" and "**arity**".

```
int get_struct_arg(csp_cell s_cell, int arg_no,
                   csp_cell *arg);
```

If the type of "**cell**" is not "t_struct" then returns "**false**". If "**arg_no**" is greater then the actual number of arguments then returns "**wrong_arg_no**" otherwise "**true**". The "**arg_no**"-th argument of the functional expression represented

by "**cell**" is assigned to "**arg**". If "**arg_no**" is zero "**arg**" is the name of "**cell**".

```
int make_int_cell(int value, csp_cell *cell);
```

"**cell**" is made to represent a fix number of value "**value**". Returns "**true**".

```
int make_float_cell(double value, csp_cell *cell);
```

"**cell**" is made to represent a float number of value "**value**". Returns "**true**" or a value meaning memory_full.

```
int make_atom_cell(char *value, csp_cell *cell);
```

"**cell**" is made to represent an atom value "**value**". Returns "**true**" or a value meaning memory_full.

```
int make_list_cell(csp_cell head, csp_cell tail,
                   csp_cell *list);
```

"**cell**" is made to represent a list with head "**head**" and tail "**tail**". Returns "**true**" or a value meaning memory_full.

```
int make_struct_cell(csp_cell name, int arity,
                     csp_cell args[],
                     csp_cell *s_cell);
```

"**cell**" is made to represent a functional expression with name "name", arity "**arity**" and arguments "**args**". Returns "**true**" or a value meaning memory_full.

```
int unify_cell(csp_cell cell1, csp_cell cell2);
```

"**cell1**" is unified with the "**cell2**". It returns "**true**" is the unification was successful, "**false**" if it was not. Any other return number means error (memory full). If the unification is "**false**" then the variable bindings are not undone that means that if this function call fails then the built in predicate must return "**false**" as well.

```
int try_unify_cell(csp_cell cell1,
                   csp_cell cell2);
```

"**cell1**" is unified with the "**cell2**". It returns "**true**" is the unification was successful, "**false**" if it was not. Any other return number means error (memory full). If the unification is "**false**" then the variable bindings are undone that means that if this function call fails then the built in predicate can continue and can succeed.

For interpreter:

```
int make_trail_note(int (*f)(), csp_cell note);
```

This function has to be used by "backtrackable" built-in predicates to remove the effect of the predicate while backtracking. If the backtrack reaches the point of calling

this predicate the function **"f"** is called with argument **"note"**. If a csp_cell is not enough to store the information needed to undo the global effects then an amount of memory has to be allocated using function **"csp_alloc"** and the pointer returned by it can be stored in a csp_cell. Don't use the "long" returned by **"csp_alloc"** as argument in **"make_trail_note"** and **"f"** because the size of **"csp_cell"** is not always equal to the size of "long" (better assign the "long" value to a csp_cell and vice versa).

For compiler:

The above function in the compiler version is replaced a pair of functions. The first of them serves for preparing the trail entry, the other one putting the trail entry into the trail stack.

```
int trail_block_register(int size, long cell_mask,
                         long proc_mask,
                         int (*f)());
```

This function prepares the trail array and it has to be called only once for every backtrackable user defined built-in predicate as a kind of initialization. The trail array is an array of csp_cells (max 32). Among these csp_cells you may want to store C pointers too. In this case pointers have to be appear as long unsigned numbers. You have to specify

size
size of the trail array (the number of csp_cells)

cell_mask
Each bit of the mask corresponds to a csp_cell in the trail entry. (The least significant bit to the 0th entry). If the Ith entry is a C pointer then the Ith bit has to be 0 otherwise 1.

proc_mask
reserved (0).

f
the address of the function which will be called on backtracking by the system. The system supplies the address of the trail array as its single argument.

This function returns an integer number which identifies the trail array to be used later in a **"make_trail_note"** call.

```
int make_trail_note(int tr_entry_id,
                    csp_cell *tr_array);
```

This function has to be called every time when you want make a trail note. The first argument is the trail array identifier got from the **"trail_block_register"** call. The second argument is the address of the trail array.

long csp_alloc(unsigned len);

This functions allocates a piece of memory of size "**len**" and returns a pointer (of type "long") which cannot be used directly to address the memory only passing as an argument to "csp_addr". Value "nil" means memory full.

char huge *csp_addr(long p);

This function returns the absolute address of the memory allocated by "**csp_alloc**" and represented by pointer "**p**".

int csp_rel(long p, unsigned len);

This function releases the memory allocated by "**csp_alloc**".

For interpreter:

int create_choice();

Used in nondeterministic built-in procedures it creates the possibility for successive alternatives of the procedure. This call has to precede any function creating structure or list and any "**unify_cell**" call. It must be followed by a "**set_choice**" or a "**destroy_choice**" function call before returning. "**create_choice**" returns "**true**" or an error (memory full).

For compiler:

int create_choice(int argno,
extb_choice_point *extb_retry);

The effect of this function is similar to the interpreter's one only the parameter passing differs. The first argument is the arity of your built-in predicate. The second argument has to be the address of a "**extb_choice_point**" type structure variable. On return "**create_choice**" fills this structure. In the compiler's case the same C function is activated at first occasion as at the subsequent choices. So the user has provide at least one extra argument in order to be able to differentiate between these two cases. The extra arguments also serve for storing the information to be passed from one activation to the next one. On PROLOG level the extra arguments have to be initialized with definite values that the first activation will recognize.

For interpreter:

int set_choice(int (* cont_func)(), csp_cell u);

When the backtrack reaches this point the interpretation continues with calling "**cont_func**" with argument "**u**". If a csp_cell is not enough to store the information needed to call the next alternative then an amount of memory has to be allocated using the function "**csp_alloc**" and the pointer returned by it can be stored in a csp_cell. Don't use the "long" returned by "**csp_alloc**" as argument in "**set_choice**" and

"cont_func" because the size of **"csp_cell"** is not always equal to the size of "long". The call of **"cont_func"** must terminate with calling **"set_choice"** if there are more alternatives or with calling **"destroy_choice"** if there are not. **"set_choice"** returns **"true"** or an error (memory full).

For compiler:

int set_choice_arg(int argnum, csp_cell cell);

Rewrites the **"argnum"**-th argument of the built-in predicate with **"cell"** on the current choice point.

int destroy_choice();

When a nondeterministic predicate doesn't want to succeed any more it has to call **"destroy_choice"** and return **"fail"**. This function always returns **"fail"**.

Never store CS-PROLOG data (csp_cells returned by interface functions) to global variables since they can be garbage collected. So any information for communication between built-in predicates has to be stored in memory allocated by **"csp_alloc"** except that the arguments and parts of arguments of nondeterministic predicates can be stored between successive calls.

8.3 Installation Of User Extensions For Interpreter

The CS-PROLOG diskettes supply three files to enable you to create your own extended interpreter:

INTERFACE.INC header file containing the global constants and structures and prototypes of the external C functions.

COPROLOG.LIB The whole CS-PROLOG interpreter in a library

CSPFACE.EXE Auxiliary program (See later)

The installation process of your own extension is the following:

1. Create your C program using the C interface function set explained above. Don't forget to include the **"INTERFACE.INC"** header file.

2. Compile your C program. For compilation use the **/AL** option of the Microsoft C compiler as CS-PROLOG does itself.

3. Put the names of all user-written built-in predicates to a file under an arbitrary name. Every name should be in separate lines. This file is called external name file.

4. Then run the auxiliary program **CSPFACE:**

   CSPFACE input_bds output_bds user_nam

   The **"input_bds"** is the binary file that is to be extended. It may be either the **COPxxxxx.BDS** file delivered with the CS-PROLOG interpreter or a previously extended binary file. **"output_bds"** is the name of the new extended binary file. The **"user_nam"** is the name of the external name file.

   The CSPFACE program performs two tasks.

   It extends the **"input_bds"** file onto the **"output_bds"** file inserting the names listed in **"user_nam"** into it.

   It generates a file named **NPUSER.C.** This file is a small C source program. You doesn't have to understand the content of this file.

5. Compile **NPUSER.C** the same way as you have compiled your c program in the 2nd step.

6. Link your interpreter from your source program object file, **NPUSER.OBJ** and the appropriate CS-PROLOG library.

   LINK @file_name

   A sample link file for **"file_name"** should be:

   ```
   obj1 obj2 ... obj_n npuser
   myexe
   llibce coprolog+
   /STACK:8192 /NOEXTDICTIONARY+
   /FARCALLTRANSLATION /PACKCODE+
   /NODEFAULTLIBRARYSEARCH;
   ```

   The file **"obj1"**, **"obj2"**, ... are the objects of your source program, **"myexe"** is the name given to the interpreter. Don't forget the options, they all are very important!

7. Invoke your extended interpreter typing

   MYEXE MYPROG.BDS

   at the DOS prompt.

8.4 Installation Of User Extensions For Compiler

The CS-PROLOG compiler diskettes supply three files to enable you to create your own extended compiler runtime system:

INTERFACE.INC header file containing the global constants and structures and prototypes of the external C functions.

CSPCOMP.LIB The whole CS-PROLOG compiler runtime system in a library

CSPCUGEN.EXE Auxiliary program (See later)

The installation process of your own extension is the following:

1. Create your C program using the C interface function set explained above. Don't forget to include the "**INTERFACE.INC**" header file.

2. Compile your C program. For compilation use the **/AL** option of the Microsoft C compiler as CS-PROLOG compiler runtime system does itself.

3. Put the names of all user-written built-in predicates to a file under an arbitrary name. Every name should be in separate lines. This file is called external name file.

4. Then run the auxiliary program **CSPCUGEN**:

   CSPCUGEN user_nam user_nam_c

   where "**user_nam**" is the external name file and "**user_nam_c**" is a filename under which of a small C source file will be generated by **CSPCUGEN**. You doesn't have to understand the content of this file.

5. Compile "**user_nam_c**" file the same way as you have compiled your C program in the 2nd step.

6. Invoke the CS-PROLOG compiler using the "**-ext**" option:

   CSPCOMP my_prog -EXT:user_nam

   where "**my_prog**" is the PROLOG program using the user-written built-in predicates and "**user_nam**" is the name of the external name file.

7. Link your compiler runtime system from your source program object file, the object file of "**user_nam_c**" and the appropriate CS-PROLOG library.

   LINK @file_name

A sample link file for "**file_name**" should be:

```
obj1 obj2 ... obj_n user_nam_c
myexe
/NOD:llibce llibcer api cspcomp+
/STACK:20480 /SEGMENT:234;
```

The file "**obj1**", "**obj2**", ... are the objects of your source program. "**user_nam_c**" stands for the object file of the small C program generated by **CSPCUGEN**. "**myexe**" will be the name of your extended runtime system. Don't forget the options, they all are very important!

8. To run your extended system you have to set a DOS environment variable:

```
SET CSPPROG=my_prog
```

where "**my_prog**" is the name of the PROLOG program to be run.

9. Invoke

```
myexe
```

at the DOS prompt.

9. The CS-PROLOG Compiler System

9.1 The CS-PROLOG Compiler

The compiler consists of two files:

- **CSPCOMP.EXE** the executable file, the compiler itself
- **CSPCOMP.BIN** internal data file for the compiler

The "**CSPCOMP.BIN**" is searched always in the same directory where the compiler is, so copying the compiler to a directory (not necessarily the working directory) copy the "**.BIN**" file as well.

The compiler generates code in an object format that is loaded dynamically by the runtime system so no linkage is necessary after compilation. The invocation of the compiler has the following form:

CSPCOMP pro_name [options ...]

The "**pro_name**" is the name of the CS-PROLOG program to be compiled without extension since "**.PRO**" extension is assumed. The generated code is stored in the file "**pro_name.LDF**" if the compilation was successful. The generated code contains only those built-in predicates that are called statically in the program (unless you specify the "**-blt**" option). The options begin with "**-**". The following options are available:

-noblt
If there are no such built-in predicate which would be called as metacalls or called by dynamic clauses added dynamically this option may be given. It ensures that only those CS-PROLOG built-in predicates are included into the generated code which are explicitly used.

-l
For efficiency purposes in the generated code two byte addressing is used. If the static code length exceeds 32K four byte addressing is needed. This can be forced by "**-l**" option. If a large program is compiled without this option and the code area exceeds the limit an error message is sent then the program has to be recompiled with "**-l**".

-opt:filename
The options of the CS-PROLOG interpreter or converter that have meaning for the compiler version (**sound, error_on_undefined, tail_recursion_opt, printer, acknowledge,** (see the interpreter manual)) can be set with this option. The "**filename**" has to

be the name of a file produced by the interpreter environment **"OPTION SAVE"** facility.

-ext:filename
This option directs the compiler that the names listed in **"filename"** are to be considered as external user-defined predicates.

The error messages of the compiler. The syntax errors are printed out in the following format:

filename(line) : error err_no : err_text

Here **"filename"** is the file name which is compiled, **"line"** is the line number where the error occurred, **"err_no"** is the number of the error, **"err_text"** is the error text. The semantic errors (e.g. simultaneously static and dynamic clause) have the same form, only the line number is omitted and the erroneous identifier is printed out.

There are several fatal errors which terminate the compilation: memory full, missing file etc. In case of a fatal error a message is printed out and the compiler exits without code generation.

9.2 The CS-PROLOG Runtime System

The CS-PROLOG compiler generates a code file that is loaded dynamically by the runtime system. This program contains the procedures needed by the generated code (e.g. the built-in predicates). The name of this program is

CSPFRAME.EXE

The system loads the generated code from a file with a fixed name:

CSP_PROG.LDF

You can change this setting an environment variable **"CSPPROG"** to the name of the **".LDF"** file generated by the compiler (the extension need not to be given) with the DOS' SET command:

SET CSPPROG=filename

The main goal of the program, i.e. the clause that is called in the beginning of the execution, is the very first clause in the source program. It has to be a clause without arguments (with zero arity). If a program uses the simulation extension predicates of CS-PROLOG, it is not necessary to use the "run" predicate to initialize the execution (as in the interpreter version) because the compiler runtime system begins its execution creating a CS-PROLOG process whose goal is the main goal of the source program.

There are several options for the runtime system. They can be given by setting another DOS environment variable **"CSPOPT"** with the DOS' SET command. The option names have to be enumerated separated by ";":

SET CSPOPT=opt1;opt2; ...

The options are the following:

medtables

mintables
The memory for the main data stacks (HEAP, STACK, TRAIL) is allocated at the beginning of the execution and cannot be extended (because of efficiency considerations). So the system has to decide what amount of memory to allocate for these stacks and the rest is available for the dynamic clauses, floats, symbols, etc., (these tables are extendible). If the total size of available memory is M bytes by default the system allocates 70% of M for the main stacks. With **"medtables"** option set this proportion is 50%, with **"mintables"** it is 30%. So if a program constructs many prolog terms and the execution is deeply recoursive the default memory sharing is good. But if the program creates many dynamic clauses, float numbers new global symbols, it is better set this option to avoid a MEMORY_FULL error.

nogc
By default the runtime system performs garbage collection if one of its tables runs out of free space. With this option the garbage collection can be disabled.

gcstat
If garbage collection is performed with **"gcstat"** option set a summary information is printed out to the screen telling the number of collected and freed items in the tables of the runtime system. It damages the current state of the screen which is not restored! (The output can be redirected to a file.)

ega
This option should be set if CS-PROLOG is run on a PC with EGA card or compatible the window scrolling is done 15 times faster. Setting this option on a CGA card causes "snowing" on screen.

10. Continuous Simulation In CS-PROLOG

The CS-PROLOG's continuous simulation extension can handle such kind of mixed models that consists of the simulation of discrete events and continuous flows. From the user's point of view the extended CS-PROLOG system is transparent, i.e. all possibilities of the basic CS-PROLOG system is still available. This chapter deals only with the added features.

Even the basic CS-PROLOG system has been able to handle discrete simulation models using discrete processes (see the chapter "Parallel Execution"). In the extended CS-PROLOG system all discrete process handling predicates still work. But beside discrete processes a single "continuous process" may be created which simulates the continuous aspect of the simulation model. The continuous part of the combined simulation model can be described by differential equations. The set of differential equation systems the user can use is the set of ordinary first order differential equation systems.

"equation" declarative clause serves for describing a differential equation system. A declarative clause syntactically is normal PROLOG fact. Its name is fixed but its arguments have to be written by the user. It is used to pass input to the simulation system. E.g. the user may insert into the CS-PROLOG program an **"equation(...)."** fact to describe a differential equation system.

Once a differential equation system is described the user can ask the system to create the continuous process using the **"cnew"** built-in predicate. The continuous process later can be deleted by the **"delete_cproc"** built-in predicate. Note that whereas only one continuous process can present at a time but the simulation model may apply more than one continuous process if they do not overlap each other in time. So the simulation program may contain several **"equation(...)."** definition. They can be scheduled either by the user (explicitly creating and deleting them) or by the system (implicitly creating and recreating them on backtracking).

The execution of the continuous process means the step by step solution of the established differential equation system as the simulation system time increases. Note that the systemtime notion of CS-PROLOG is assigned to the independent variable of the differential equation system and the dependent variables of the differential equation system can be regarded as the state variables of the simulation model. In every time moment the current value of the state variables can be asked using the **"state_variable"** built-in predicate. If for some reason backtrack occurs and the system time decreases then the state variables will get back their previously computed values. Depending on the behavior of the simulation model the system time may move forward and backward but in every time moment the state variables will contain the right values.

Sometimes it is necessary to wait until a given condition on state variables becomes true. E.g. to determine when a

state variable will reach a constant boundary or when two state variables will become equal. It can be done using the "**wait_for_condition**" built-in predicate.

The differential equation definitions may contain "parameters". They can be set either statically in the "**equation(...).**" clause or dynamically using parameter handling predicates "**set_param**" and "**get_param**". If the program changes the value of a parameter during runtime the effect of the change is done at the system time moment when the "**set_param**" action was issued. Parameters can be set backtrackable way using the "**set_value_b**" built-in predicate which means that on backtracking the parameter setting is undone to the previous value.

The user has several opportunities to observe the behavior of the state variables.

Using the "**plot**" declarative clause any combination of the state variables can plotted against the system time in a coordinating system on the screen. Phase portrait of two arbitrary state variable can be plotted as well.

The history of a simulation run can be documented on the printer using the "**record**" declarative clause and printing, drawing and dumping built-in predicates.

10.1 Declarative Clauses

Declarative clauses appear in the user's program as PROLOG facts. (A fact is a normal PROLOG clause which has no body). They serve for storing information in their arguments and make them available to the simulation system. Normally they will never executed explicitly, only the system "reads" them. Their names are fixed and the user has to take care of not mix them with other clause names in the program.

There are three kind of declarative clauses:

- equation
- plot
- record

Each kind of declarative clauses may form a partition (as every normal PROLOG clause does). Since the first argument of all of them should be a name of a differential equation system (see later the description) all declarative clauses of the same name belong to the same differential equation system.

If more than one "**equation**" declarative clauses are present in the program with the same name (for example "**equsys**") then multiple alternatives of the same differential equation system are defined. If later the "equsys" differential equation

system is assigned to the continuous process using the **"cnew"** built-in predicate then its first alternative is applied. If during the execution the simulation model fails and the backtrack reaches the **"cnew"** predicate then in nondeterministic way **"cnew"** chooses the second alternative of the **"equsys"** differential equation system. Further backtracks will reassign the third, fourth etc. alternatives. If there is no more alternative to choose **"cnew"** will fail itself.

In case of **"plot"** and **"record"** declarative clauses multiple alternatives with the same name are allowed but only the first is used by the system.

10.1.1 Equation Definition

equation(Name,<eq_init>,<eq_def>).

equation(Name,<eq_init>,<eq_def>,stiff).

equation(Name,<eq_init>,<eq_def>,stiff,
Error_limit).

Defines an differential equation system with initial value conditions. **"Name"** must be a constant. It will be the name of the differential equation system. **"<eq_init>"** and **"<eq_def>"** describe the initial value condition and the differential equation system respectively. The optional fourth argument must be the constant **"stiff"** if it is present. It marks the differential equation system to be solved with a special solver algorithm rather than the general one. It is the user's responsibility to decide what kind of solver is appropriate for a differential equation system. It is advisable to use the default solver which will be efficient in most cases. However, if the execution of the continuous process aborts because of the stiffness of differential equation system the user can try to use the **"stiff"** solver algorithm. Nevertheless the **"stiff"** algorithm will surely inefficient for the most non-stiff differential equation systems. Even the stiff solver algorithm can be further refined using the fifth **"Error_limit"** argument. If it is omitted the error limit of the stiff solver algorithm is assumed **0.01** . If the error limit is decreased then the solution will be more precise and the computation will be slower.

The syntax of the initial value condition and the differential equation system definition is the following:

```
<eq_init> =     var    := expr   |
              (  var_1  := expr_1
              {, var_i  := expr_i} ... )

<eq_def> =      var'   := expr   |
              (  var_1' := expr_11
              {, var_i' := expr_1i} ... ) {: cond_1}

          {;  (  var_1' := expr_k1
              {, var_i' := expr_ki} ... ) {: cond_k}
          } ...
```

Metavariables "**var...**" should be different PROLOG names. They will be the symbolic names of the state variables. Note that these symbolic names has no meaning outside the scope of the differential equation system definition and the state variable referencing declarative clauses or built-in predicates. E.g. "**y1**" may refer to state variable in the "**equation(...).**" declarative clause but in the normal PROLOG environment will appear as an ordinary literal "**y1**". The user should avoid to confuse the notion of the state variable with the normal PROLOG variable.

The number of equations in the differential equation systems is maximized in **10**. The order of state variables on the left side of the initial value conditions and the differential equations have to be the same.

Metavariables "**expr...**" should be usual arithmetical expressions built from numbers and symbolic names of the state variables, subexpressions, parameters and special names (see below) connected with the usual arithmetic operators. (See the description of the "**is**" built-in predicate).

Beside the state variable names the user can use in "**expr...**" symbolic expression names and symbolic parameter names.

Symbolic expression names serves for reducing the complexity of the right side of an equation. The actual subexpression can be assigned to a symbolic expression name using the "**set_expression**" built-in predicate.

Symbolic parameter names stand for a numerical value in the definition. Their value can be set using the "**set_param**" built-in predicate during runtime. All parameters of a differential equation system must have been already set when the differential equation system is used by a "**cnew**" predicate. The number of parameters in a differential equation system is maximized in **20**.

The independent variable of the differential equation system (the 'time' variable) can be referenced using the fixed name "**t**" . There are two more predefined constants for "**pi**" and "**e**" .

Metavariables **"cond..."** should be logical expression built-from arithmetical expressions defined above connected with the usual relation operators (see the chapter "Comparison predicates") and the usual logical operators (**"not", "and", "or", "xor"**). If conditions are used then instead of one differential equation system a set of differential equation systems is defined. Each element of the set must have the same number of equations and equations must refer to the same state variables in the same order. The number of alternatives is maximized in 8. The total number of equations in a set of differential equation systems cannot be greater than **32.**

The use of the conditions enables the user to describe such differential equation systems the right side of which may vary depending on the given conditions. During the evaluation the solver continuously checks the condition system. If there is at least one condition which is true then the solver algorithm chooses the differential equation system associated with the first true condition. Otherwise a runtime error will be generated.

Examples:

```
equation(sinus_cosinus, ( y1  :=   0,
                          y2  :=   1 ),

                        ( y1' :=  y2,
                          y2' := -y1 ) ).

equation(spring,    ( y1  :=   0,
                      y2  :=   1 ),

                    ( y1' :=  y2,
                      y2' := -y1 + d*y2 ) : t <= 8;

                    ( y1' :=  y2,
                      y2' := -y1 + d/2 * y2 ) ).

equation(stiff_eq, ( x1  :=   1,
                     x2  :=   0,
                     x3  :=   0 ),

                   ( x1' := -0.1 * x1,
                     x2' :=  0.1 * x1 - 1.e8 * x2,
                     x3' := 1.e8 * x2 ), stiff).
```

10.1.2 Plotting On The Screen

It can be helpful if the user can follow the graph of the solution function during the differential equation system evaluation. For this purpose the continuous extension of CS-PROLOG environment provides a tracing tool. The tracing tool, if it is activated, draws the graphs of the required solution functions on the screen step by step as the function values are computed by the solver algorithm. The user can

determine which functions have to be plotted and which area of the coordinating system has to appear on the screen. Different functions are plotted with different colors. There are two way to plot curves. In the first case user can determine the edges of a rectangular by their coordinates. The system automatically normalizes the rectangular to be fit properly on the full screen. In the second case the user can determine a horizontal zone. In that case the screen acts like an oscilloscope screen. The graphs of the solution functions are drawn from left to right. If they reach the right border of the screen it is cleaned up and the drawing continues from the left border and so on. If the differential equation system is changed on backtrack then the affected graphs are redrawn on the same screen. So backtrack may cause to plot a set of graphs of the same solution function. The user can activate the tracing tool by including the a **"plot"** declarative clause into the program.

plot(Name,Statvar_list,V_intval,H_intval).

plot(Name,Statpair_list,V_intval,H_intval,phase).

Its first argument is the identifier of the declaration. This identifier should be the same name as one of the **"equation"** declarations. The second **"Statvar_list"** argument is a PROLOG list. Its elements should be symbolic names of either state variables of the differential equation system referred by **"Name"** or parameters. The state variables and/or parameters listed here are drawn on the screen. The number of curves on the screen is maximized in **6**. The third **"V_intval"** argument must be a list containing two numbers. They determine the vertical interval of the plot. Depending on whether the user wants to apply the rectangular or horizontal zone plotting method the fourth **"H_intval"** argument can be either a list of two numbers (as in the vertical case) or a number. In the latter case this number determines during the oscilloscope-like plotting what size of slice of the logical horizontal axis have to fit on the screen.

If the user wants to plot phase portraits he should use the second form of **"plot"**. The **"Name"** is the same as above. The second argument must be a list of two element's sublists. The elements of the sublists can be symbolic names of either state variables of the differential equation system referred by **"Name"** or parameters. The number of phase portraits on the screen is maximized in **6**. The meaning of the third and fourth argument is the same as above in the rectangular plotting case. The fifth argument must be the constant name **"phase"**.

10.1.3 Documentation On The Printer

After the simulation run the user may want to print out or draw out the graphs or to dump the values of the solution functions. ("Printing" is done in character mode, "drawing" in bitmap mode.)

To obtain an output document on the printer the user should do the following actions:

- announce the request of recording the computed function values using the **"record"** declarative clause.

- print out the documentation of the recorded values using built-in predicates.

Note that the user can only print out the documentation of a continuous process when it has finished its task.

The recording request is announced by the presence of the **"record"** declarative clause in the program.

record(Name,Statvar_list,Step_size).

Its first argument is the identifier of the declaration. This identifier should be the same name as one of the **"equation"** declarations. The second **"Statvar_list"** argument is a PROLOG list. Its elements should be symbolic names of either state variables of the differential equation system referred by **"Name"** or parameters. The state variables and/or parameters listed here are recorded by the system during execution. The third argument determines the step size of the recording. The impact of that declaration is that the solver algorithm puts a record containing the values of listed state variables and/or parameters into a file. It puts a record to the file in every step from the start time until the end time of the continuous process. Backtracks may cause that there is more than one record referring to the same systemtime moment. But the printout generating built-in predicates discussed below make a preprocessing first in order to discard the superfluous records and obtain the right set of function values. In every multiply recorded case the last record is retained. So the remained set of records will be identical with the solution functions accepted by the simulation run.

10.2 Continuous Simulation Built-in Predicates

10.2.1 Continuous Process Handling

In CS-PROLOG a discrete process is characterized by its goalsequence, its name and the time interval within which it must terminate its task. We can imagine the continuous process in a quite similar way. The only difference is that the goalsequence is replaced by the name of a differential equation system. While the discrete process performs its task by executing its own goalsequence then the continuous does it by solving the equation system assigned to it. A further restriction is that only one continuous process can exist at a

given systemtime moment. It does not mean that only one differential equation system can be present in the program (alternative **"equation(...)."** declarations) or only one continuous process can be created during the simulation run but the user should plan their lifetime in a such way that only one of them can be active at a given systemtime moment, i.e. the user cannot create another continuous process (**"cnew"**) until the actual continuous process is not deleted (**"delete_cproc"**).

cnew(Name,Process_name,Start_time,End_time)

cnew(Name,Process_name,Start_time)

cnew(Name,Process_name)

The continuous process is created by the **"cnew"** predicate. The first argument is the name of the differential equation system. **"cnew"** must satisfy the following conditions: An **"equation(Name,...)."** declaration with **"Name"** as its first argument must be present in the PROLOG's database and it must be syntactically correct. **"cnew"** cannot define such a continuous process the lifetime of which (the time interval between **"Start_time"** and **"End_time"**) overlaps the lifetime of a previously created continuous process. The start time must not be earlier then the current systemtime. Otherwise an error message is generated. If everything is correct the scheduler initiates the continuous process. It becomes 'created-but-inactive' and waits until the systemtime reaches its **"Start_time"**.

"cnew" makes possible to define different equations on backtracking. The **"equation(...)."** declaration partition may contain several clauses (facts). Some of them may have the same name in their first argument. These clauses form a 'sub-partition' in the **"equation"** partition. These subpartitions play an important role on process creation during backtracking. They serve for alternative equation assignment. Let's have the following example:

```
equation(abc, eq_abc).

equation(xyz, eq_xyz_1).

equation(xyz, eq_xyz_2).

..., cnew(xyz, xyz_proc, 0, 10), ...
```

When **"cnew"** is performed first time it initiates the **"xyz_proc"** with the **"eq_xyz_1"** differential equation system and succeeds. If casually the backtrack reaches **"cnew"** the scheduler tries to find another **"equation"** declaration with the same name and reinitiates the **"xyz_proc"** with the **"eq_xyz_2"** equation system and succeeds again. A further backtrack would not find any new alternative and would continue the backtrack before **"cnew"**. In other words **"cnew"** tries all possible alternative differential equation system declarations with the same name in a non-deterministic way.

delete_cproc(Process_name)

"**Process_name**" must be the name of the currently active continuous process. "**delete_cproc**" removes the active continuous process from the simulation system.

10.2.2 Differential Equation System Evaluation

Since the independent variable of the differential equation system is assigned to the systemtime of the CS-PROLOG their evaluation is done automatically by the system. Whenever the user asks the value of a state variable then the appropriate value of the solution function will be returned. When the systemtime is increased the differential equation system is solved automatically. When a backtrack causes the systemtime to be decreased the solution function is rolled back and the state variables will contain their previous values. If the scheduler finds a choice point anywhere in the program and it can go forward again the solving of the differential equation system continues from that system time moment.

After its creation the continuous process gets into 'created-but-inactive' state i.e. it waits until the systemtime reaches its start time. Obviously it solves the differential equation system only during its lifetime. If the systemtime reaches the end time of the continuous process then it is deleted from the system and the execution of discrete processes continues. Any later reference to its state variables is erroneous. Of course a later backtrack may cause the continuous process to reappear again. Conversely in that case when the backtrack continues beyond its start time it becomes 'created-but-inactive' again. If the backtrack reaches even the creation point the "**cnew**" will work as we discussed above.

10.2.3 Asking For State Variables

state_variable(Stat_var,X)

The user can ask the value of a state variable using the "**state_variable**" built-in predicate. At the systemtime moment when the "**state_variable**" is executed the continuous process must be active. The "**Stat_var**" must be the symbolic name of a state variable of the currently active differential equation system. "**state_variable**" always succeeds and unifies the current value of the given state variable with **X** at the current systemtime moment.

10.2.4 Waiting For Conditions

wait_for_condition(Cond)

Sometimes the user wants to determine a systemtime moment when the state variables satisfy a given condition first time. For example when the value of a state variable reaches a constant numeric value or when two graphs of state variables cross each other or any other logical expression becomes true first time. This can be done using the **"wait_for_condition"** built-in predicate. The **"Cond"** may be a complex logical condition using state variables, parameters, numerical constants and usual logical and relation operators. When a discrete process calls this predicate it will be suspended until the given logical condition becomes true. In other words it means an implicit **"hold"** suspension of the caller discrete process for an unknown time interval. Note that if the systemtime reaches the end time of the continuous process and there is a discrete process suspended by a **"wait_for_condition"** call it means a deadlock situation and the scheduler forces to backtrack the simulation run in order to find out another alternatives. No more than **20 "wait_for_condition"** request may be active at a time.

10.2.5 Parameter Handling

Sometimes the user wants to slightly modify the differential equation system during the simulation run. For example he wants to change the value of a constant in the equation definition. For that purpose continuous extension of the CS-PROLOG introduces the notion of 'parameters'. The notion of parameters is very similar to the notion of 'values' in the basic CS-PROLOG. The difference is that the parameters can only store numerical values and symbolic parameter names can appear on the right side of a differential equation definitions while 'values' cannot. Parameters are global numerical variables which can be set and get by parameter handling predicates anywhere and any time. Since they are global variables one parameter can be used in several equation definitions. Note that parameters can be set not only during the lifetime of the continuous process but every time during the simulation run.

Parameters can be set using one of the following built-in predicates.

set_param(Par_name, Numerical_constant)

set_param_b(Par_name, Numerical_constant)

The only difference between them is that the first predicate is non-backtrackable, the second one is backtrackable. They set the **"Par_name"** parameter to the **"Numerical_constant"** value. If the **"Par_name"** parameter is

referred by an equation and this equation is currently being solved the calling of the **"set_param"** or **"set_param_b"** has an immediate effect on the differential equation solution. From that systemtime moment when the set was encountered the solver algorithm continues the evaluation with the newly set parameter value until a further setting does not take place.

get_param(Par_name, X)

The **"get_param"** predicate serves for getting the current value of a parameter.

set_expression(Expr_name,Expr).

"Expr_name" must be a PROLOG name. **"Expr"** must be an arithmetical expression similar to the right side of the differential equation definitions (see **"equation(...)."** declarative clause). **"set_expression"** associates the **"Expr"** arithmetical expression with the name **"Expr_name"**.

10.2.6 Printer Handling

Obviously the user wants to print out the result when the continuous process has been finished. It can be achieved by one of the following built-in predicates.

print_graph(Name,Pair_list,Printer_type,From,To)

print_graph(Name,Pair_list,Printer_type)

Its first argument should be the same name as one of the **"equation"** declarations. Furthermore it is desirable that a **"record"** declaration should have been inserted to the program with the same name otherwise **"print_graph"** will not find the file of the recorded values and it generates an error message. The second argument is PROLOG list of two element's sublists. The first element of every sublist is state variable which should have already appeared in the second argument of the appropriate **"record"** declaration. If it did not appear it cannot draw and it is ignored. The number of state variables and/or parameters to print is maximized in **6**. The second element of every sublist must be an PROLOG atom. Usually it is a single character. It serves for determining that character by which the graph of that state variable will be printed out. If the length of the given atom is greater then one its first character is used. The third argument must be either the constant name **"epson"** or anything else. If **"epson"** is used then **"print_graph"** uses EPSON printer specific control codes and produces more pretty output than in the general case. The forth and fifth argument (if exist) determine two systemtime moment within which the solution functions will be printed out. If they are omitted the whole lifetime of the continuous process is assumed.

draw_graph(Name, Statvar_list, From, To)

draw_graph(Name, Statvar_list)

This predicate can be used only if the user's configuration has an EPSON FX-... printer or compatible. Its first, third and fourth argument have the same meaning as in the case of **"print_graph"**. The second argument is quite similarly contains a PROLOG list of state variable names. These state variables should satisfy the same conditions. This predicate works in the same way as the previous one but it draws the solution functions more precisely and smoothly using the bitmap mode of the printer.

The user has the opportunity to print out the result of the differential equation evaluation in a table form.

print_record(Name, Statvar_list, From, To)

print_record(Name, Statvar_list)

Its arguments have the same meaning as above. It does the same task as **"print_graph"** but it prints out the value of solution functions in a table format. Since its page breaking method is identical with the **"print_graph"** one's it is a good idea to print out both kind of output and examine them side-by-side.

11. Index

D

E

F

G

H

I

K

L

M