CS-PROLOG

Version 3.25.

MULTIPROCESSOR MODEL

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 multiprocessor model on any kind of transputer network with an IBM XT/AT under DOS as host machine.

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)

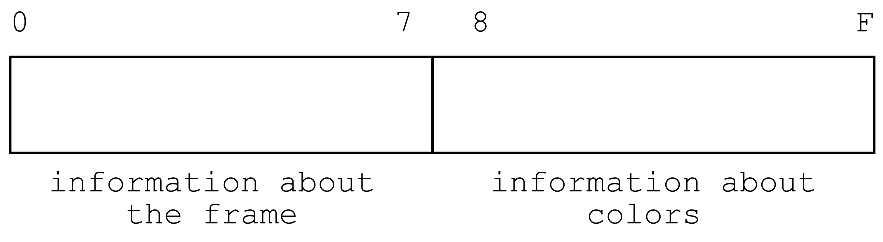
- number of rows and number of columns
  (1 - 25, 1 - 80)

- 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

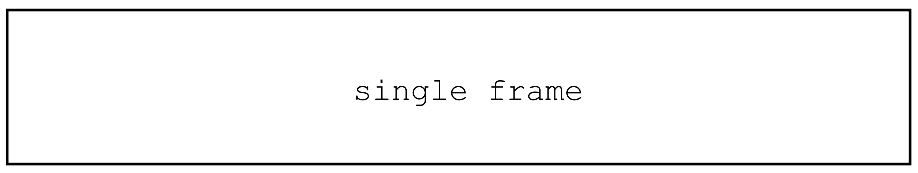
The attribute is an integer. This number is used as a 16 bit pattern:



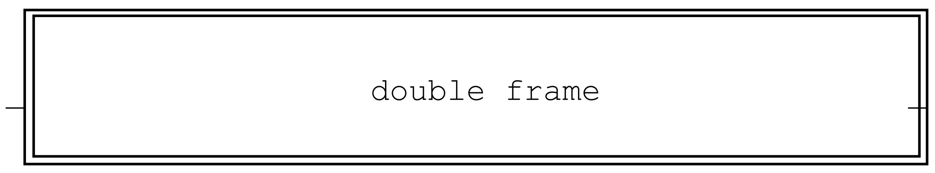
The left side byte can have any of the following 5 values:

0 There is no frame on the window

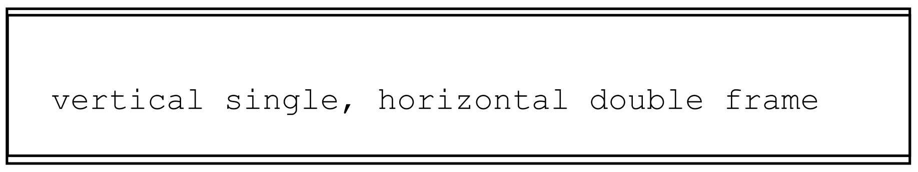
1



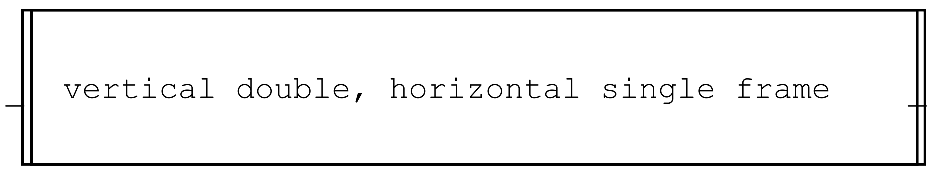
2



3

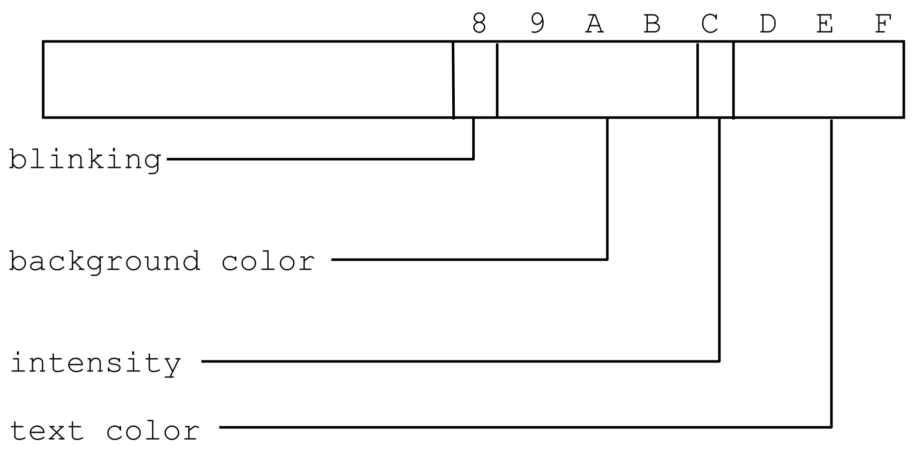


4



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)

Non implemented in the multiprocessor environment.

load_system(C)

Non implemented in the multiprocessor environment.

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

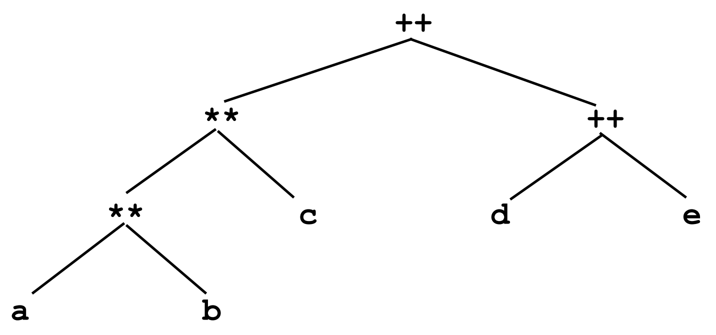
The priority must be an integer in the range between -3000 and 3000.

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

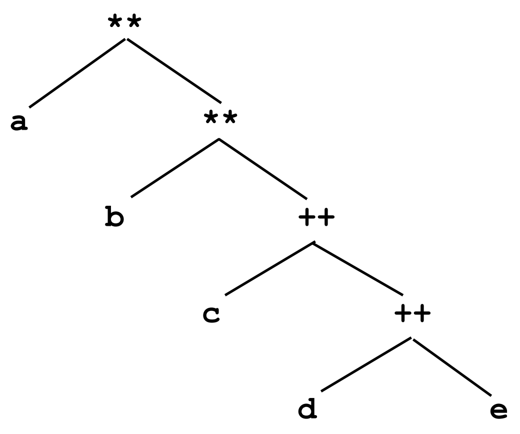
can be illustrated in the following parenthesized and tree format:

((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

CS-PROLOG graphics are not yet implemented in the multiprocessor environment. Future versions will provide them.

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 multiprocessor environment processes can be created in separate processors. If no more than one process is assigned to each processor then processes can run parallel in real time. However when some of the processes share the same processor more or less the scheduling algorithm of the monoprocessor version is used on the given processor.

5.2 Scheduling On A Single Processor

Since only one process can be "active" at a time on a single processor other processes are in waiting state. Scheduler allows the active process run until the process reaches any of predicates causing its suspension or until it solves its task. Then another activable process is chosen until every process finishes its task on that processor or they all fail.

Since the forward execution of processes results a sequentialized order of execution, backtracking is executed backward in this path. This means that everything is undone until the last choice point even through message sending operations. The last choice point is either a Prolog choice point or a waiting operation or process exchange.

5.3 Distributed Control Mechanism

Let's assume further that no more than one process is assigned to any processor in a multiprocessor environment. In this case a distributed control mechanism assures the synchronization of the processes.

To understand the distributed control mechanism of CS-PROLOG imagine that an ordinary CS-PROLOG interpreter works on each processor. Until the processes do not use the **"new"** and **"send"** built-in predicates (called communication points) these interpreters are independent and work in parallel. In forward step crossing a communication point increases the possible degree of parallelism either by creating a new process or sending a message for a waiting process.

The crucial point is met in the distributed control mechanism when a process crosses a communication point during backtracking. This can happen in two essentially different cases:

- in failure backtracking
- in dead-lock backtracking

Failure backtracking occurs in a processor **P** when the active process has failed. As a matter of fact this kind of backtracking appears in ordinary sequential Prolog interpreters too. However in the distributed environment meanwhile processor **P** is doing failure backtrack the interpreters of other processors can execute either forward steps or failure backtracking too. If processor **P** crosses a communication point during failure backtracking the effect of the communication point should be undone. If the created or addressed process is also on processor **P**, this requirement is automatically fulfilled. This is true, because due to the one processor available, the created or addressed process could not get the control over the processor yet, and the predicates new and send (see later) being deterministic and backtrackable the process creation and the message sent is undone or deleted. However if the communication point affects another processor this effect should be undone in a distributed way. For this purpose the so-called anti-message is sent by the communication point during backtracking. The anti-message forces the interpreter of the receiver processor to backtrack to the point where the original message was consumed.

Dead-lock backtracking is needed when the system reaches a point where at least one process is waiting for a message, while the others are in finished state or they are also waiting for messages and no transient message in the multiprocessor network. This situation is called global dead-lock. In global dead-lock one process is chosen (called forced process) for executing dead-lock backtracking while the others are waiting for reactivation by the forced process. The dead-lock backtracking mechanism terminates when the backtracking process finds an alternative path in its search tree which contains a communication point. This communication point gives the chance the system to leave the global dead-lock state either by creating a new process or by sending a message.

Notice that in deadlock backtracking only one processor is active executing the backtracking and all the others are passive while in failure backtracking any processor can be active by executing either forward steps or failure backtracking. One can generally say that the main difference between failure backtracking and deadlock backtracking is in their parallelism. While the former is parallel in nature the latter is sequential.

5.4 The Algorithm of the Supervisor

In CS-PROLOG we distinct the local scheduler of processors from the global scheduler of multiprocessing system, however the local scheduler actually works as a part of the global scheduler.

Some basic definitions:

(1) Active process:

the process currently executed on the processor (a process being either in Running Forward state, or Running Backward state or Transient Waiting state)

(2) Waiting process:

the process that has not been activated yet or is to be reactivated at a given time (because it is suspended for a certain time interval)

(3) List of waiting processes:

the waiting processes assigned to the processor

(4) Terminated process:

the process that has successfully executed its goals

(5) List of terminated process:

the terminated processes assigned to the processor

(6) Frozen process:

the process that was stopped during its execution by a **"delete_process"** call

(7) List of frozen processes:

the frozen processes assigned to the processor

(8) Blocked process:

the process that is waiting for a message

(9) List of blocked processes:

the blocked processes assigned to the processor

(10) Backtracked process:

the process backtracked before it has successfully executed its goal

(11) List of backtracked processes:

the backtracked processes assigned to the processor

5.4.1 Local Schedulers

The algorithm of local schedulers implemented on a single processor is the following:

```
(1)  If there are some waiting processes,

     then the local scheduler chooses the first one
          of them, it will be the active process that
          will get in Running Forward state and (2),

     else (7)

(2)  The active process remains in Running Forward
     state

     until

          (a)  it is terminated (all of its
               predicates succeeded

          (b)  it reaches a wait_for-type predicate
               call

          (c)  it has to backtrack because of lack of
               matching alternatives

          (d)  it is frozen by another process

          (e)  it reaches a delete_process-type
               predicate call

     In case of (a) and (d) (3).

     In case of (b)

          if there is a proper message

               then takes such a message and
                    continues according to (2)

               else the process gets in the list of
                    blocked processes and (3)

     In case of (c) backtracking, (5).

     In case of (e) the process sends a freezing
          command, and (2).
```

(3) The processor examines the commands arriving from the scheduler according to the following viewpoints, then deletes the considered command. (The order is unimportant if several cases are examined at the same time.)

```
If

(a)  a Pulling Back command has arrived to one
     of the processes of any list,

     then

          if the process has taken the message,

               then takes the process off the
                    list, it will be the active
                    process, and backtracks
                    according to (4),

               else the local scheduler sends a
                    Backtrack Continuing command
                    to the Pulling Back process,
                    and (3).

(b)  a Message Invalidating command has arrived
     to one of the processes of any list,

     then

          if the process has taken the message,

               then takes the process off the
                    list, it will be the active
                    process, and backtracks
                    according to (6),

               else (3)

(c)  a Process Creation Invalidating command has
     arrived (see 5b.) to one of the processes
     of any list,

     then backtracking to the point of creation

(d)  a Restarting command has arrived (because
     of 5g.),

     then puts the processes of the given
          Transient Waiting list and the
          backtracked processes to the list of
          waiting processes

(e)  a Backtrack Continuing command has arrived
     to one of the processes of Transient
     Waiting state,

     then it continues the backtracking
          according to (5)
```

(f) a Backtracking command has arrived (because of global dead-lock),

then

if there is a backtrackable process,

then takes such a process off the list, it will be the active process, and backtracks according to (5),

else the dead-lock remains and after sending a 'there is no backtrackable process' message to the global scheduler the processor goes to (3)

(g) no message has arrived

then (7)

(4) It is the same as (5), however if the backtracking reaches the point where the message sent by the Pulling Back process was taken, then it sends also a Backtrack Continuing command to the process that pulled it back originally.

(5) Backtracking: the backtracking process continues its backtracking

until

(a) it reaches a send_b-like predicate call

(b) it reaches a new_b-like predicate call

(c) it reaches a delete_b-like predicate call

(d) it backtracks before its first predicate call

(e) it reaches a wait_for-like predicate call

(f) it reaches a wait_for_dnd-like predicate call

(g) it can't choose a new alternative at a choice point

```
In case of (a)

     if the backtracking is global passive,

          then puts the process on the Transient
               Waiting list, sends a Pulling
               Back command to the process
               taking the message, and (3)

          else local or global active
               backtracking, sends a Message
               Invalidating command, and
               continues backtracking, (5)
```

In case of (b) the algorithm is similar to that of (a), with necessary modifications.

In case of (c) sends an Invalidating of Freezing command to the frozen process, and (5).

In case of (d) puts the active process on the list of backtracked processes, and (3).

In case of (e) it remains there until the message is invalidated (because of backtracking or dead-lock situation), then puts the process on the list of blocked processes, and (3).

```
In case of (f),

     if   there is a message that hasn't been
          tried yet

          then takes the next message, sends a
               Restarting command and gets in
               Running Forward state according
               to (2),

          else puts the process on the list of
               blocked processes, and (3)
```

In case of (g), choosing a new alternative the currently backtracked active process gets in Running Forward state, the scheduler sends a Restarting command to each transient waiting and backtracked process, then (2).

(6) Continuous backtracking without choosing a new alternative, according to (9), while we reach the invalidated message or the point of taking "**new_b**", meanwhile if we reach a "**send_b**" or a "**new_b**" call, then send further invalidating commands. If we reach the point of taking the message, then (5) (e-f) will be applied.

(7) If there are processes that got matching messages,

then these processes are taken off the blocked list, and are put on the waiting list in the same order, (1),

else (8).

(8) If there are frozen processes that got an Invalidating of Freezing command (an other process backtracked before the **"delete_process_b"** predicate call freezing it,

then these processes are put on the list of waiting processes, and (1)

else (3)

(9) It is the same as (5), however it considers only (a-d), and referencing to (5) is substituted by referencing to (9).

Note: if the scheduler gets to an endless loop (3)-(7)-(8) on all processes, this is the case of global dead-lock, that should be realized by the global scheduler.

5.4.2 Global Scheduling

The processors get a serial number. The "rootprocessor", which is nearest to the host machine, gets the number 1.

The algorithm of the global scheduler is the following:

I. Wait until one if the following cases is true:

(a) each processor gets in Terminated state

(b) each processor gets in Backtracked state

(c) the multiprocessing system gets in a dead-lock situation

In case of (a) the CS-PROLOG program has succeeded.

In case of (b) the CS-PROLOG program has failed.

In case of (c) distributed backtracking, (II.).

II. Choose the processor with the lowest serial number, which is not in Backtracked state, and it has to get a command to backtrack (3). If there is no such process, the program has failed.

5.5 Simulation In A Multiprocessor Environment

The method of handling time is different from the monoprocessor system since there is no global system time. Instead every process has its own local time. When a process waits for a message at a given point it has to know which processes can send messages to this point. Before it consumes a message it has to wait until every process sends its message to this point. Then it has to consume the message sent with the smallest time stamp. The local time has to be modified using this time stamp. This mechanism ensures that messages will be consumed in the order of their sending time.

It is up to the user to achieve the above tasks (as waiting for all messages and setting the right local time using the built-in predicate **"advance"** for the latter purpose).

The actual release of the multiprocessor version of CS-PROLOG supports only this so called conservative approach. This means that simulation programs written in the single processor version generally does not run correctly in this release.

To make correctly executed simulation programs the user has to ensure the correct synchronization of the processes in the simulation time.

The next release will contain the so called Time Warp algorithm to ensure full compatibility between the old monoprocessor version and the new multiprocessor version and to make possible the so called optimistic distributed simulation.

5.6 Process Manipulating Predicates

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

Creates a new process with goal **GOAL**, name **NAME**, starting time **START**, and termination time limit **END**. Each process should have a distinguished **NAME**. If START is missing or is a variable it is set to the local time of the caller process. If **END** is missing or is a variable it is set to 10^50. If **PROCESSOR** is given it identifies the processor where the new process will be executed. A processor identifier is a number between **1** and **N** where **N** is the number of processor in the current configuration. If this argument is missing the new process will be executed on the caller's processor. The "**new**" predicate is backtrackable, i.e. during backtracking through "**new**" a so called anti-message is sent to the processor where the original message created the new process. As a result of the incoming anti-message the process is removed from the system. (This predicate is deterministic and backtrackable.)

Implementation hint! From implementation reasons, during the execution of an application program the **name** of a process is assigned to the processor that the process was the first time allocated to. It means if a process disappears from the system either due to backtracking or successful termination or explicit deletion the name of the process remains still assigned to its processor. So a further recreation of any process with the same name (even if the **GOAL** differs) must be directed to the same processor.

new_unb(GOAL)
new_unb(GOAL,NAME)
new_unb(GOAL,NAME,START)
new_unb(GOAL,NAME,START,END)
new_unb(GOAL,NAME,START,END,PROCESSOR)

The "**new_unb**" predicate is deterministic and non-backtrackable, i.e. during backtracking through "**new_unb**" no anti-message is sent and therefore the created process is never deleted.

Restriction: the created process must run on another processor than the process creating it.

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 (broadcast) it can be done using an unbound variable as **PROC_LIST.** The local time of the sender process is time-stamped to the message. The "**send**" predicate is deterministic and backtrackable which means that during backtracking through "**send**" a so called anti-message is sent to each process of **PROC_LIST** forcing those processes to backtrack to the point where the original message was consumed and as a result of the incoming anti-message the original message is removed.

send_unb(MESS,PROC_LIST)

The "**send_unb**" predicate is deterministic and non-backtrackable which means that during backtracking through "**send_unb**" no anti-message is sent to the processes of **PROC_LIST.**

wait_for(MESS)
wait_for(MESS,T)

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 and a matching message is sent to the waiting process. If the second argument is given "**wait_for**" unifies **T** with the time-stamp of the message M. (This predicate is deterministic and backtrackable.)

wait_for_dnd(MESS)
wait_for_dnd(MESS,T)

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.)

wait_for_unb(MESS)
wait_for_unb(MESS,T)

Non-backtrackable version of "**wait_for**". The message consumed by it can not be put back to the system during backtracking.

Restriction: the sender and the receiver processes should be on different processors.

advance(T)

The execution of the active process is NOT suspended for **T** time units. (As it is for "**hold(T)**" in the monoprocessor version.) Rather the local time of the caller process is incremented by **T**. In the multiprocessor environment there is no global system time each process has its own local time updated by the "**advance**" or the "**wait_for_..**" built-in predicates.

active_process(AP)

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

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**.

local_time(T)

Unifies **T** with the local time of the caller process.

set_local_time(T)

Sets the local time of the caller process to T if the local time is less than **T**.

own_processor(OP)

Unifies **OP** with the identifier of the processor that runs the caller process.

max_processor(MP)

Unifies **MP** with the maximal number of processors in the network. The processor identifiers are in the range of (1,MP). The maximal value of MP is 254.

p_make_ground(X)

Unifies each unbound variable in the term **X** with a constant which is unique in the whole multiprocessor system. Notice that "**make_ground**" uses constants which unique only on the caller processor.

terminate_system

Forces all processes to immediately terminate and the system considers them as successfully terminated processes therefore the whole system successfully terminates.

5.7 Important notes

Important note! If the "**!**" (cut) has effect on a "**wait_for**" predicate it causes an error! Avoid constructions like the following:

```
a :- b , !.

b :- wait_for(c).
```

The use of "**run**" predicate to start the CS-PROLOG program is not needed any more in the multiprocessor version. To start your program type only your "**Goal**" (single goal only).

All database handling predicates have a "**local_**" version. The principle of CS-PROLOG claims that processes can only communicate via messages. However the ordinary database handling predicates allow the communication of processes running on the same processor through the database. In some cases this feature may be advantageous however extremely dangerous. For example after reconfiguration of your system two processes communicating through database can run on different processors preventing them in communicating through the database. In order to avoid the undisciplined communication between processes one can use the **"local_"** versions of the database handling predicates. These exclude the possibility of communicating through database even between processes running on the same processor.

The correctness of the distributed deadlock mechanism becomes unpredictable if process communication is done through the database.

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.

```
      > Entry Ö------------------------Ì  + Success
        -----> °                        °  ----->
               °   Predicate definition  °
    - Failure °                        °  < Re-Entry
        <----- Û------------------------ì  <-----
```

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.
X	The trace skips the current window for one step. In a multiprocessor environment it makes possible to slow down the given processor or to concentrate only on one processor.
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 processor in the system the trace screen can be split into several windows. The trace list of different processes running on the same processor is displayed in the same window but the trace list of different processes running on different processors is displayed in different windows.

Using the trace window facility the windows are generated for processors instead of processes. For each processor the one-window trace facility is available in a separate window. On each window there is a headline containing the following information:

[processor_id] process_name T=local_time

The user can specify the number of trace windows in the **Setup** menu.

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 processors can be directed into different windows (see the description of the interactive trace). You can specify here the processors to be traced in different windows, using a browser. Maximum 8 windows can be used so max. 8 processors can be selected in this browser. Use the 'space' key to select or deselect a processor. The processors are represented by their serial number. The default state is to trace in one window the processor number 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

7.1 Bank Robbery Problem

File name "BANK.PRO".

This program is the basic CS-PROLOG demonstration program. In the multiprocessor environment the fifth arguments in the "**new**" built-in predicates are set to 1 and 2 denoting that dick is running on the 1st processor and jim is running on the 2nd processor.

To run the program type:

problem.

7.2 Maze Problem

File name "MAZEPGR.PRO".

This program finds the path in a maze from the starting point to the exit point. The program runs on four processors each representing the north, east, south, west directions respectively. The work of the processors are shown in the screen by four figures representing the same maze but in different states. Red paths show the current search path of the given processor.

To run the program type:

problem.

7.3 Eight Queens Problem

File name "QQ8GR.PRO".

This program represents the CS-PROLOG solution for the 8-queens problem. It can run on any number of processors but exploits maximum 8 processors.

To run the program type:

problem.

or

problem(N).

In the first case all processors (up to 8) of your system will be used to solve the problem. In the second case the first N processors of your system will be used to solve the problem. The solutions are displayed by graphics on eight different chess-boards. The solution may be time consuming. After solving the problem the running time of the program is written in seconds in a red window. The execution time of the problem for different numbers of processors can be compared by using different N parameters.

7.4 Pub Problem

File name "PUBCONS.PRO".

The short description of the problem is the following. There is a bar with two entrances. In each 5 time units a customer is coming through the first door and in each 3 time units a client is coming through the second door. In total there are 3 customers coming through the first and 5 customers coming through the second door. The barman (mixer) needs 4 time units to serve a customer. After receiving their drinks customers spend 15 time units in the bar.

To run the program type:

problem.

As an output the programs lists on the screen the termination time of the processes.

7.5 Parser Problem With Multiple Solution

There are seven different solutions elaborated for the same parser problem. The description of the problem and the explanation of the different solution algorithms can be found in a companion file in printable format:

parexer.doc

The grammar of a natural language parser is defined by the following rules:

(1) sentence -> noun_phrase, verb_phrase

(2) noun_phrase -> determiner, noun

(3) verb_phrase -> verb, noun_phrase

(4) verb_phrase -> verb

The task is to decide about a series of list of words if they construct sentences in the given grammar or not.

The sequential algorithm is realized by the

lparsesw.pro

program.

To run the program type:

problem(N,D).

where N is the number of word_lists to be processed (maximum 20) and D represents the execution time of the nodes of the Parse Tree. D can be any positive integer. The Parse Tree is drawn on the screen. The process of parsing is shown on the screen by recolouring those nodes of the Parse Tree that are currently activated. In a Result window the result of parsing each word_list is displayed. After solving the problem the running time of the program is written in seconds in a red window. By modifying the value of D the execution time can be influenced.

The parallel algorithm based on the processor farm concept is realized by the

lparspfw.pro

program.

To run the program type:

problem(N,D,P).

where N is the number of word_lists to be processed (maximum 20) and D represents the execution time of the nodes of the Parse Tree. D can be any positive integer. P is the number of processors to be used to solve the problem (maximum 4). A Worker process is allocated to each processor and a Worker window is displayed on the screen for each Worker process. Whenever a Worker process is activated by a message the corresponding Worker window is recoloured by the color of the given processor. After finishing the program execution each Worker window contains a number representing the number of nodes of the Parse Tree executed by the corresponding Worker process. The running time is also displayed as mentioned above.

The next solution can be found in

lparipfw.pro

which is an improved version of the previous program where a simple processor manager is used by the master process to balance the work of the processors.

To run the program type:

problem(N,D,P).

A static AND-parallel solution can be achieved by the

lparsspw.pro

program.

To run the program type:

problem(N,D).

Again the whole Parse Tree is drawn on the screen like in case of lparsesw.pro. The process of parsing is shown on the screen by recolouring those nodes of the Parse Tree that are currently activated. Nodes executed on the same processor are recoloured with the same color. Notice that the noun_phrase and verb_phrase processes are executed on different processors. After finishing the program execution the noun_phrase and verb_phrase windows contain a number representing the number of nodes of the Parse Tree executed by the corresponding processors. The running time is also displayed as mentioned above.

A pipeline algorithm is realized by the

lparspip.pro

program.

See explanation in parexer.doc file.

To run the program type:

problem(N,D).

This program also illustrates how to organize the program to save memory by backtracking after the processing of each word_list. Backtracking empties the stacks of Prolog making it possible to execute long programs too.

An improved pipeline algorithm is realized by the

lparspi2.pro

program.

See explanation in parexer.doc file.

To run the program type:

problem(N,D).

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 3L Parallel C compiler V 2.0. For the multitransputer environment see also the "Installation" chapter.

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.

9. Installation

9.1 Hardware & Software Requirements

The multitransputer version of CS-PROLOG system is currently implemented only on transputer network connected to an IBM XT\AT as a host machine. You need:

1. IBM PC/AT (or compatible) as host, CGA or EGA, 1.2M floppy drive, hard disk, 640K memory, DOS 3.3 or later
2. Transputer board(s) (T414 or T800 at least 1 Mbyte per transputer)
3. 3L Parallel C compiler, linker, configurer

9.2 Interpreter System Installation

The distribution diskettes of CS-PROLOG interpreter system has the following structure:

```
DISK #1

     root directory:
          common system files for both T414 and T800
          transputers

          T414 directory:
               T414 specific system files

          T800 directory:
               T800 specific system files

          EX directory:
               CS-PROLOG examples

          CONFIG directory:
               hardware description samples for
               installation

DISK #2

     root directory:

          T414 directory:
               T414 specific system files for C
               interface option of CS-PROLOG

               EX directory:
                    CS-PROLOG example for C interface
```

T800 directory:
T800 specific system files for C interface option of CS-PROLOG

EX directory:
CS-PROLOG example for C interface

The algorithm of installation is the following:

1. Make sure that you have enough space on your hard disk for the CS-PROLOG system : 2 Mbytes and cca 300K per transputers.

2. Create the directory called **"csprolog"** for the CS-PROLOG system on your default hard disk using the following DOS commands :

```
md \csprolog
cd \csprolog
```

3. Copy the root directory of the DISK #1 to the **"csprolog"** directory.

```
copy a:\*.*
```

4. If you have T414 transputers copy the T414 directory of DISK #1 to the **"csprolog"** directory.

```
copy a:\t414\*.*
```

5. If you have T800 transputers copy the T800 directory of DISK #1 to the **"csprolog"** directory.

```
copy a:\t800\*.*
```

6. Create the hardware description file of your transputer configuration using a text editor. The **"config"** directory of the DISK #1 contains two possible description files for either 2 or 4 transputer connection:

   - mway2.cfg
   - mway4.cfg

   It would be useful to simply rewrite one of them according to your exact configuration. The hardware description file should contain:

   - the identifiers of processors
   - the existing physical wires between the processor links

   E.g.

```
<edit> myconfig.cfg
```

7. Having your own description file call "**cspbuild**" batch in order to generate your own application.

   T414 case :

   ```
   cspbuild /T4 myconfig.cfg csprolog.app
   ```

   T800 case :

   ```
   cspbuild /T8 myconfig.cfg csprolog.app
   ```

   The list of available options of "**cspbuild**" batch can be obtained calling it without parameters.

   ```
   cspbuild
   ```

   Hint! There are some machines where the "**config**" program can not terminate properly but the generated output file is correct. In that case you might stop the configurer after a couple of minutes by pressing CTRL-C.

   If you are planning to use the C interface option of CS-PROLOG system later you have to rebuild your application in a slightly different manner (see Using C interface).

8. If this generation was successful your application is ready to use. It is advisable to create another working directory for your prolog programs. In that case you should either insert the "**csprolog**" directory into the PATH list or copy the "**csp.bat**" file into the working directory. Create your working directory and enter into it.

   E.g.

   ```
   md \work
   cd \work
   copy \csprolog\csp.bat
   ```

9. Once you are in the working directory copy the CS-PROLOG examples from the "**ex**" directory of DISK #1 into the working directory:

   ```
   copy a:\ex\*.*
   ```

   and then invoke the CS-PROLOG system typing

   ```
   csp
   ```

   Now you can run any example CS-PROLOG program.

9.3 Installation Of The C Interface In Interpreter

The CS-PROLOG system allows the user to write his or her own built-in predicates and append them to the standard built-in predicate set of CS-PROLOG. The full description of C interface routines can be found in the chapter "External C Interface". The multitransputer considerations are the following:

The extension routines have to be written in 3L Parallel C but they MUST NOT use the parallel features of 3L Parallel C.

The user should keep in mind that his or her extension will appear on every transputer but they are absolutely independent. So there is no way to communicate through global variables neither.

The algorithm of generating the extended application is the following. In the current example we are going to assume a T414 transputer so the specific file names and extension often refer to the T414 transputer. If you are using a T800 transputer change every "**4**" character to "**8**".

The first phase of the algorithm should be executed only once for prepare the creating of a user extended CS-PROLOG application.

1. Enter into the "**csprolog**" directory

   cd \csprolog

2. If you have T414 transputers copy the following files from DISK #2 to the "**csprolog**" directory.

   copy a:\t414\*.*

3. If you have T800 transputers copy the following files from DISK #2 to the "**csprolog**" directory.

   copy a:\t800\*.*

4. Rerun the "**cspbuild**" batch by adding a further option to its command line. This option will cause the necessary files not to be deleted during the generation process. These files serve as input to the configuration of the user extended application.

   ```
   cspbuild /T4 /N /L\csprolog
            myconfig.cfg
            dummy.app
   ```

5. The previous step has created the necessary configurator's input file for the forthcoming configuration process in the **"\csprolog"** directory. Copy the following file into your working directory.

   ```
   copy \csprolog\$cspb$.cfg
        \work\my_ext.cfg
   ```

6. Edit **"my_ext.cfg"** using a text editor.

7. Change every occurrence of the

   "\csprolog\csprolog.b4"

   pattern into

   "\work\my_ext.b4"

   using your text editor's replace facility. Hopefully you will find as many such patterns as many transputers you have. Note that in the current example a T414 transputer set is assumed. If you have T800 transputer both pattern must end with **".b8"**.

8. Quit your text editor.

Now your system is ready to create the user's extended application of CS-PROLOG. The second phase of the creation process should be executed whenever you want to modify something inside one of your own built-in predicates.

1. Create or modify your extension routines using a text editor.

   E.g.

   my_ext.c

   From **"my_ext.c"** collect the function names that are going to be used as a built-in predicates into a file

   my-ext.nam

2. Compile your C source using the 3L Parallel C compiler.

   ```
   t4c /Fomy_ext.bi4 my_ext.c
   ```

Your "**my_ext.c**" necessarily refers to the include file "**interfac.inc**". It resides currently in "**\csprolog**" directory. You can access it by one of the following ways:

```
copy \csprolog\interfac.inc
```

or using the following include directive in your C program:

```
#include "\csprolog\interfac.inc"
```

3. Run the "**cspface**" program

```
afserver -:b \csprolog\cspface.b4
         \csprolog\csprolpp.btp
         my_ext.btp
         my_ext.nam -:o 1
```

4. Compile the generated "**npuser.c**" using the 3L Parallel C compiler.

```
t4c /Fonpuser.bi4 npuser.c
```

5. Create the appropriate link file using a text editor. In the current example the link file

   "my_ext.lt4"

   should look like this:

```
my_ext.bi4    (user's extension compiled
              binary file)
npuser.bi4
\csprolog\csprolog.li4
              (CS-PROLOG interpreter's
              library)
\tc2v0\sacrtlt4.bin
              (3L Parallel C library)
\tc2v0\taskharn.t4
              (3L Parallel C library)
```

6. Link all binary files into a new task using 3L Parallel C linker

```
linkt @my_ext.lt4,my_ext.b4
```

7. Configure the previously linked task and some standard system task into a new application containing the user's built-in predicates using 3L Parallel C configurator. Assume that the input file of the configurator "**my_ext.cfg**" have been created (see above).

```
config my_ext.cfg my_ext.app
```

8. Run your extended application by typing the following lines at the DOS prompt or by including them into an own batch file:

```
@echo off
\csprolog\cspcanc
\csprolog\cspmsup \csprolog\cspmsup.bds
afserver -:b my_ext.app my_ext.btp
         %1 %2 %3 %4 %5 >csprolog.his
```

9.4 Compiler System Installation

The distribution diskettes of CS-PROLOG compiler system has the following structure:

```
DISK #1

     root directory:
          common files for both T414 and T800
          transputers of runtime system

          T414 directory:
               T414 specific system files

          T800 directory:
               T800 specific system files

          EX directory:
               CS-PROLOG examples

          CONFIG directory:
               hardware description samples for
               installation

DISK #2

     root directory:
          The compiler itself to be run under DOS
```

The algorithm of installation is the following:

1. Make sure that you have enough space on your hard disk for the CS-PROLOG system : 2 Mbytes and cca 300K per transputers.

2. Create the directory called **"cscomp"** for the CS-PROLOG system on your default hard disk using the following DOS commands :

```
md \cscomp
cd \cscomp
```

3. Copy the root directory of the DISK #1 to the "cscomp" directory.

```
copy a:\*.*
```

4. If you have T414 transputers copy the T414 directory of DISK #1 to the "**cscomp**" directory.

   copy a:\t414\*.*

5. If you have T800 transputers copy the T800 directory of DISK #1 to the "**cscomp**" directory.

   copy a:\t800\*.*

6. Create the hardware description file of your transputer configuration using a text editor. The "**config**" directory of the DISK #1 contains two possible description files for either 2 or 4 transputer connection:

   - mway2.cfg
   - mway4.cfg

   It would be useful to simply rewrite one of them according to your exact configuration. The hardware description file should contain:

   - the identifiers of processors
   - the existing physical wires between the processor links

   E.g.

   <edit> myconfig.cfg

7. Having your own description file call "**cspbuild**" batch in order to generate your own application.

   T414 case :

   cspbuild /T4 myconfig.cfg csprun.app

   T800 case :

   cspbuild /T8 myconfig.cfg csprun.app

   The list of available options of "**cspbuild**" batch can be obtained calling it without parameters.

   cspbuild

8. If this generation was successful your application is ready to use. It is advisable to create another working directory for your prolog programs. In that case you should either insert the **"cscomp"** directory into the PATH list or copy the **"csp.bat"** file into the working directory. Create your working directory and enter into it.

   E.g.

   ```
   md \work
   cd \work
   copy \cscomp\csp.bat
   ```

9. Before the first compilation install the CS-PROLOG compiler and the delivered examples executing the following:

   Insert Disk #1 in drive A:

   ```
   copy a:\ex\*.*
   ```

   Insert Disk #2 in drive A:

   ```
   copy a:*.*
   ```

9.5 The CS-PROLOG Compiler

The compiler consists of two files:

- **MCSCOMP.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:

```
MCSCOMP 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.

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.

9.6 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 invocation of this program is

```
CSP filename [options]
```

at the DOS prompt.

Note that during the multitransputer execution a special host support program must reside in the host PC's memory. In the case of normal termination the system automatically removes the host support program. However in an abnormal termination case the support program may still remain in the host PC's memory. You can remove it manually typing

```
CANCEL
```

at the DOS prompt.

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 on the command line of the **"CSP"** batch:

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.

9.7 Installation Of The C Interface In Compiler

The current version of the CS-PROLOG compiler in the multitransputer environment does not provide the C interface possibility. Future versions will contain it.

10. Index