DyALog Primer

August 1997

Eric de la Clergerie

INRIA Rocquencourt
Domaine de Voluceau
78150 Le Chesnay Cedex, FRANCE

This manual corresponds to DyALog version 1.9.0, 12 November 2001

Introduction

This manual describes the system DyALog developped at the "Institut National de Recherche en Informatique et Automatisme" [INRIA] in France. DyALog is used to compile tabular executable from Logic Programs and Definite Clause Grammars. While working for standard Prolog-like programs programs, DyALog is essentially helpful to build efficient parsers for highly ambiguous and recursive grammars as found in Natural Language Processing.

Indeed, tabular executables keeps traces of sub-computations in a table in order to get computation sharing and loop detection. They also ensure computation completness and give the possibility to test different evaluation strategies.

Notational conventions

When referring to keyboard characters, printing characters are written thus: `a`, while control characters are written like this: ^A. Thus ^C is the character you get by holding down the ⟨CTL⟩ key while you type `c`. Finally, the special control characters carriage-return, line-feed and space are often abbreviated to ⟨RET⟩, ⟨LFD⟩ and ⟨SPC⟩ respectively.

When introducing a built-in predicate, we shall present its usage with a *mode spec* which has the form *name(arg, ..., arg)* where each *arg* denotes how that argument should be instantiated in goals, and has one of the following forms:

:ArgName The argument should in the program correspond to a goal.

+*ArgName*
: The value of the argument should not be a variable.

-*ArgName* The value of argument should be a variable.

?ArgName
: No constraint on this argument.

In the context of some directives, we shall need the following notation: Predicates in Prolog are distinguished by their name *and* their arity. The notation *name*/*arity* is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `concatenate/3` specifies the predicate which is named "concatenate" and which takes 3 arguments.

More generaly, a *predicate spec* may be

name/*arity*
: the elementary form

`[`*elem_form*`,...]`
: a Prolog list of elementary forms

pred_spec1`,`*pred_spec2*
: a comma-separated list of predicate specifications

`dcg(`*pred_spec*`)`
: a *dcg predicate spec* to refer to DCG predicates

1 Installation

1.1 Obtaining DyALog

DyALog is available as a source package or as a binary rpm at `ftp://ftp.inria.fr/INRIA/Projects/Atoll/Eric.Clergerie/DyALog/`.

The WEB page `http://atoll.inria.fr/~clerger/` proposes documentation as well as an access to the distributions.

1.2 Supported Machines

The current version of DyALog can only run under Linux on i*86 architectures.

1.3 Installation

More detailled explanations are given in '`INSTALL`' when installing from a source distribution.

1.3.1 Configuring DyALog

1. Run '`./configure`' to generate the various Makefiles.

1.3.2 Installing DyALog

3. Type '`make`'.
4. Type '`make install`' to export the program binaries and librairies.
5. [optional] Type '`make clean`'.

2 Using DyALog

Unlike most Logic Program evaluators, DyALog has no toplevel, being designed to compile parsers.

The main command in the DyALog package is `dyacc` which is a PERL script used to compile programs.

This command uses `dyalog` to compile Prolog files (`.pl`) into DyALog Assembler files (`.ma`), and `dyam2asm` to convert `.ma` files into machine specific assembler (`.s`). The C compiler `gcc` is then called to build object files (`.o`) and link them.

2.1 Dyacc

The PERL script `dyacc` is a frontend to `dyalog`, `dyam2asm`, and `gcc`. An analysis of the command line is done to correctly forward the options to the different commands.

To use dyacc, issue the shell command:

`dyacc [`*options* `|` *files*`]... [--` *cc-options-and-files* `]`

where the possible options are

'`-c`' Compile or assemble the source files, but do not link. The compiler output is an object file corresponding to each source file.

'`-dev`'

To be used when `DyALog` is not installed (development mode)

'`-I` *path*'

Add '`path`' to the set of pathes used by `dyalog` to find files.

'`-ma`'

Compile DyALog source files, but do not call `dyam2asm` or `gcc`.

By default, `dyacc` makes the object file name for a source file by replacing the suffix '.pl' with '.ma'. Use '-o' to select another name.

'`-o` *file*'

Place output in '`file`'. This applies regardless to whatever sort of output `dyacc` is producing.

'`-parse`'

Set option -parse for `dyalog`

'`-pl-ext` *suffixe*'

Specify an extra '`suffixe`' for Prolog files

'`-save-temps`'

Keep intermediate files (.ma and .s) but do not transmit the option to gcc.

'`-v`'

Print (on standard error output) the commands executed to run the stages of compilation.

'`-x` *lang*'
: Specify explicitly the '`language`' for the following input files (rather than choosing a default based on the file name suffix). This option applies to all following input files until the next '-x' option. Possible values of '`language`' are 'pl', 'ma', and 'none' (to reset).

'`--`'
: Mark the end of DyALog options. Everything on the right is passed to `gcc` and not considered as a `dyacc` option.

2.2 Dyalog

To use DyALog, issue the shell command:

`% dyalog -a [`*options* `|` *files*`]`

where the possible options are

'`-version`'
: Version information

'`-f` *filename*'
: Load the program file *filename*. '`-f`' may be ommited when *filename* does not start with '-'.

'`-I` *path*'
: Add *path* to the search path list of DyALog. The same effect can be achieved using the environment variable `DYALOG_PGMPATH`.

'`-parse`'
: Compile grammar rules considering a parsing done from a database of tokens.

'`-use` *filename*'
: Add *filename* to the list of modules to be imported. The same effect can be achieved using a directive `require/1`.

'`-res` *filename*'
: Extend the compiler by loading the resource file *filename*. The same effect can be achieved using a directive `resource/1`.

The command `dyalog` also inherits all options available to DyAlog executables: they should take place before -a.

2.3 Dyam2asm

This command is used to convert DyALog Assembler files into machine assembler.

Its syntax is

`% dyam2asm [option...]` *input_file*

'`-help`'
: Show some help and exit

'`-o` *filename*'
: Name the output file (otherwise use the standard output)

'`-version`'
: Print version number and exit

2.4 DyALog executables

All DyALog executables accept the following options:

```
%> <dyalog_exe> [options | files] [-a args]
```

'`-h`' Display some help and exit.

'`-db` *filename*'
: Load '`filename`' as a database. '`-db`' may be ommited when '`filename`' does not start with '`-`'.

'`-forest`' Display the shared forest at the end of the execution

'`-fcount`' Display the number of possible derivations per answer.

'`-slex` *string*'
: Use *string* as the string to parse

'`-flex` *file*'
: Use *file* as a character file to parse

'`-v` *kind*' Display trace information relative to *kind*, which should belong to `dyam`, `share`, `index`, or `all`.

'`-a` *args*' All *args* are collected in a Prolog list of symbols and accessible by the executable through the builtin `argv/1`.

By default, a filename on the command line is loaded as a database.

2.5 An example

We illustrate the use of dyalog on a small recursive example that will loop with standard Prolog systems.

```
% cat pgm.pl
q(f(f(a))).
q(X) :- q(f(X)).
?-q(X)    % the query must be inside the file
% dyacc pgm.pl -o pgm
% ./pgm
Answer:
    X = f(f(a))
Answer:
    X = f(a)
Answer:
    X = a
```

3 Behind the screen

DyALog uses

1. Logical Push-Down Automata [LPDA] as operational devices to describe various resolution and parsing strategies for logic programs
2. Dynamic Programming techniques to break LPDA computations in elementary subderivations that are combinable and sufficiently compact to be tabulable.

This chapter presents briefly the theoretical background behind DyALog and some internal details about its implementation.

3.1 LPDA

Logical Push-Down Automata are a natural extension of the Push-Down Automata. They may be non-deterministic. The main difference is the use of unification for transition application.

We consider three basic kinds of transitions:

1. Push
2. Swap
3. Pop

3.2 Dynamic Programming

3.3 Compilation Process

Given a set of clauses (and eventually a query), the compilation process builds some code of an Abstract Machine and a set of objects that encapsulate this code. The resulting code is either emulated or emitted toward a C file.

3.3.1 From programs to LPDA

3.3.2 From LPDA to Abstract Machine Code

3.3.3 Emitting C code

The emitting phase emits the compiled code to a C file. Futrhermore, some additional code needed to build terms, objects and to run initialization is also emitted.

3.4 Execution

4 Syntax

DyALog tries to comply with the standard syntax of Prolog, with extensions to handle hilog terms and typed features terms. On the contrary, the DyALog reader may miss some obscure points of the standard.

4.1 Terms

4.1.1 Standard terms

A standard term is either a simple term (an integer, a character, a symbol or a variable as defined in most Prolog) or a term `f(`*t1*`,...,`*tN*`)` where *ti* is a term. Note that floats are not yet implemented and that chars are not implemented as a subset of integers (but as a proper type).

4.1.2 Immediate unification

DyALog performs immediate unification at reading time when encountering infix operator `::/2`. Immediate unification is generally used to assign in a single step a variable for a whole structure and variables for its sub-structures.

For example:

```
p(X::f(Y,Z)) :- check(Y),check(Z),q(X).
```

is a shortcut for

```
p(f(Y,Z)) :- check(Y),check(Z),q(f(Y,Z)).
```

Mutiple immediate unification may take place at the same time, which is sometimes usefull in conjonction with feature terms.

4.1.3 Operators

Infix, prefix or postfix operators with precedence are allowed in DyALog and are just syntactic sugar for standard Prolog term. For instance `t+q` is equivalent to `+(t,q)`.

The declaration of new operators is possible through the usual directive `op/3`.

4.1.4 Feature Terms

It is possible to associate to a symbol (say `employee`) a list of features (say `[name,job,salary]`). When building a term based on `employee`, it is not necessary to assign explicitely and in order a value for all its features because the missing values will be filled by new anonymous variables. For instance, the feature term `employee{salary=>6000,name=>john}` is equivalent to the term `employee(john,_,5000)`. Note the use of enclosing `{}` instead of enclosing `()` to mark feature terms.

To associate a feature table to a symbol, use the directive `features/2`.

```
:-features(employee,[name,job,salary]).
```

It is also possible to use Typed Feature Structure, following the same syntax.

4.1.5 Enumeration variables

DyALog provides Enumeration Variables, i.e. variables that may take their values from some defined enumeration. For instance, the term `X::tense[present,past]` denotes a variable `X` with value in the sub-enumeration `[present,past]` of some user-defined enumeration `tense`. Note the use of enclosing *[]* instead of enclosing *()*.

To associate an enumeration to a symbole, use the directive `finite_set/2`.

```
:-finite_set(tense,[present,past,futur]).
```

It is also possible to define sub-enumeration using the directive `subset/2`

```
:-finite_set(letter,[a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,
                     q,r,s,t,u,v,w,x,y,z ]).
:-subset(voyel,letter[a,e,i,o,u,y])
```

Enumeration are restricted to at most 30 elements. These elements should be ground objects. Enumeration variable may be unified with variables, enumeration variables based on the same enumeration and with elements of their enumeration.

4.1.6 Hilog terms

Hilog terms are an extension found in some Prolog evaluators (XSB among others) that gives a flavor of (pseudo) higher order very practical to build meta-predicates or closures.

The key idea is to consider that a sequence `t (t1,..,tN)` (with a ⟨SPC⟩ between the terms) is equivalent to `apply(t,t1,...,tN)`.

The middle space can be removed when there is no ambiguity, for instance when *t* is an integer, a char, a variable, a compound term or a symbol declared as being hilog.

In case of ambiguity between an operator-based expression or an hilog expression, the operator-based expression will be chosen. For instance, `- (a+b)` represents the term `-(a+b)` and not `apply(-,a+b)`.

One can force the hilog interpretation of a symbol by using the directive `hilog/1`.

The following program illustrate the use of hilog terms to build meta-predicates.

```
closure(R)(X,Y) :- R(X,Y).
closure(R)(X,Y) :- R(X,Z),closure(R)(Z,Y).

:-hilog(r).
r(a,b).
r(b,c).
```

4.2 Programs

4.2.1 Clauses

4.2.2 Definite Clause Grammars

A clause of a Definite Clause Grammar is introduced with the binary predicate `-->/2` and closed by a dot. Lists in position of predicates in the clause denote terminals to be scanned. Scanning is done either from a Prolog list or from a token database (implementing a Finite State Automata).

The following program implements reverse with a Definite Clause Grammar.

```
reverse(X,Y) :- phrase(rev(Y),X,[]).
rev([]) --> [].
rev([X|Y]) --> rev(Y),[X].
```

4.2.3 Directives

Directive clauses are conjunctions of directives introduced par the unary predicate `:-/1` and close by a dot mark.

```
:-include('foo.pl'),op(300,xfx,[hello]).
```

5 Built-In Predicates

Not every standard built-in predicate found in most Prolog evaluators are avaliable in DyALog and will not be because DyALog is mostly devoted to build parsers.

5.1 Input / Output

Input / Output in DyALog is done through streams that can associated either to a file, a string or a device.

A stream can be connected to a filename or UNIX file descriptor for input or output by calling the predicate `open/3`.

The possible formats of a stream are:

n
: A stream connected to some file. *n* is an integer.

symbol
: where *symbol* has been aliased to a stream using `add_stream_alias/2`. Note that `user_input`, `user_output`, and `user_error` are by default aliased to the UNIX `stdin`, `stdout`, and `stderr` streams.

Shell-like expansions of filenames is provided.

5.1.1 Reading-in Programs

None.

5.1.2 Input and Output of Terms

`read(`*?Term*`)`
`read(+`*Stream*`,`*?Term*`)`
: The next term, delimited by a full-stop (i.e. a `.`, possibly followed by layout text), is read from *Stream* and is unified with *Term*. The syntax of the term must agree with current operator declarations. If a call `read(`*Stream*, *Term*`)` causes the end of *Stream* to be reached, *Term* is unified with the term `eof`. Further calls to `read/2` for the same stream will then fail, unless the stream is connected to the terminal.

`read_term(`*?Term*`,+`*Vars*`)`
`read_term(+`*Stream*`,`*?Term*`,+`*Vars*`)`
: Same as `read/1-2` with a third argument `+`*Vars*. This argument is unified with a list of *Name*=*Var* pairs, where each *Name* is an atom indicating the name of a non-anonymous variable in the term, and *Var* is the corresponding variable.

`write(`*?Term*`)`
`write(+`*Stream*`,`*?Term*`)`
: The term *Term* is written onto *Stream* according to current operator declarations.

`writeln(`*?Term*`)`
`writeln(+`*Stream*`,`*?Term*`)`
: Same as `write/1-2` except that a newline is send,

`display(`*?Term*`)`
: The term *Term* is displayed *onto the standard output stream* (which is not necessarily the current output stream) in standard parenthesized prefix notation.

`writeq(`*?Term*`)`
`writeq(+`*Stream*`,`*?Term*`)`
: Similar to `write(`*Stream*`,`*Term*`)`, but the names of atoms and functors are quoted where necessary to make the result acceptable as input to `read/2`, provided the same operator declarations are in effect.

5.1.3 Character Input/Output

`get_char(?`*C*`)`
`get_char(+`*Stream*`,?`*C*`)`
: *C* is the next character read from *Stream* (or by default, from stream `user_input`).

`put_char(+`*C*`)`
`put_char(+`*Stream*`,+`*C*`)`
: Character *C* is output onto *Stream* (or by default, onto stream `user_output`).

5.1.4 Stream IO

`open(+`*FileName*`,+`*Mode*`,-`*Stream*`)`
: If *FileName* is a valid file name, the file is opened in mode *Mode* (invoking the UNIX function `fopen`) and the resulting stream is unified with *Stream*. *Mode* is one of:

  `read` Open the file for input.

  `write` Open the file for output. The file is created if it does not already exist, the file will otherwise be truncated.

  `append` Open the file for output. The file is created if it does not already exist, the file will otherwise be appended to.

`close(+`*Stream*`)`
: If *Stream* is a stream the stream is closed.

`absolute_file_name(+`*RelativeName*`,?`*AbsoluteName*`)`
: This predicate is used by all predicates that refer to filenames for resolving these. The argument *RelativeName* is interpreted as a filename according to the filename syntax rules (see Section 5.1 [Input / Output], page 11). If the specified file is found (possibly with a '`.pl`' extension), *AbsoluteName* is unified with the full path name of this file.

`current_input(`*?Stream*`)`
: Unify *Stream* with the current input stream. The current input stream is also accessed by the C variable `SP_curin`.

`current_output(`*?Stream*`)`
: Unify *Stream* with the current output stream. The current output stream is also accessed by the C variable `SP_curout`.

`set_input(+`*Stream*`)`
: Set the current input stream to *Stream*.

`set_output(+`*Stream*`)`
: Set the current output stream to *Stream*.

`flush_output`
`flush_output(+`*Stream*`)`
: Flush all internally buffered characters for *Stream* to the operating system.

`from_alias_to_stream(+`*Alias*`,?`*Stream*`)`
: Unify *Stream* with the stream aliased to *Alias*.

`add_stream_alias(+`*Stream_or_Alias*`,+`*Alias*`)`
: Alias the stream given by *Stream_or_Alias* with *Alias*.

5.1.5 Socket IO

None.

5.1.6 DEC-10 Prolog File IO

The DEC-10 prolog IO predicates are available with the library '`dec10`':

`see(+`*File*`)`
: File *File* becomes the current input stream. *File* may be a stream previously opened by `see/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `see/1` associated with the same file already, then it becomes the current input stream. Otherwise, the file *File* is opened for input and made the current input stream.

`seeing(`*?FileName*`)`
: *FileName* is unified with the name of the current input file, if it was opened by `see/1`, with the current input stream, if it is not `user_input`, otherwise with `user`.

`seen`
: Closes the current input stream, and resets it to `user_input`.

`tell(+`*File*`)`
: File *File* becomes the current output stream. *File* may be a stream previously opened by `tell/1` or a filename. If it is a filename, the following action is taken: If there is a stream opened by `tell/1` associated with the same file already, then it becomes the current output stream. Otherwise, the file *File* is opened for output and made the current output stream.

`telling(`*?FileName*`)`
: *FileName* is unified with the name of the current output file, if it was opened by `tell/1`, with the current output stream, if it is not `user_output`, otherwise with `user`.

`told`
: Closes the current output stream, and resets it to `user_output`.

5.2 Arithmetic

Arithmetic is performed by built-in predicates which take as arguments *arithmetic expressions* and evaluate them. An arithmetic expression is a term built from numbers, variables, and functors that represent arithmetic functions. At the time of evaluation, each variable in an arithmetic expression must be bound to a non-variable expression. An expression evaluates to a number, which may be an *integer*.

Only certain functors are permitted in an arithmetic expression. These are listed below, together with an indication of the functions they represent. X and Y are assumed to be arithmetic expressions. Unless stated otherwise, the arguments of an expression may be any numbers.

`+(`*X*`)` The value is X.

`-`*X* The value is the negative of X.

X`+`*Y* The value is the sum of X and Y.

X`-`*Y* The value is the difference of X and Y.

X`*`*Y* The value is the product of X and Y.

X`//`*Y* The value is the *integer* quotient of X and Y.

X `mod` *Y* The value is the *integer* remainder after dividing X by Y, i.e. `integer(`*X*`)-integer(`*Y*`)*(`*X*`//`*Y*`)`.

X`/\`*Y* The value is the bitwise conjunction of the integers X and Y.

X`\/`*Y* The value is the bitwise disjunction of the integers X and Y.

X`#`*Y* The value is the bitwise exclusive or of the integers X and Y.

`\(`*X*`)` The value is the bitwise negation of the integer X.

X`<<`*Y* The value is the integer X shifted left by Y places.

`X>>Y` The value is the integer X shifted right by Y places.

`abs(`*X*`)` The value is the absolute value of X.

`min(`*X*`,`*Y*`)`
: The value is the lesser value of X and Y.

`max(`*X*`,`*Y*`)`
: The value is the greater value of X and Y.

Arithmetic expressions, as described above, are just data structures. If you want one evaluated you must pass it as an argument to one of the built-in predicates listed below. Note that it only evaluates one of its arguments, whereas all the comparison predicates evaluate both of theirs. In the following, X and Y stand for arithmetic expressions, and Z for some term.

Z `is` *X* X, which must be an arithmetic expression, is evaluated and the result is unified with Z.

X `=:=` *Y* The numeric values of X and Y are equal.

X `=\=` *Y* The numeric values of *X* and *Y* are not equal.

X `<` *Y* The numeric value of *X* is less than the numeric value of *Y*.

X `>` *Y* The numeric value of *X* is greater than the numeric value of *Y*.

X `=<` *Y* The numeric value of *X* is less than or equal to the numeric value of *Y*.

X `>=` *Y* The numeric value of *X* is greater than or equal to the numeric value of *Y*.

5.3 Comparison of Terms

These built-in predicates are meta-logical. They treat uninstantiated variables as objects with values which may be compared, and they never instantiate those variables. They should *not* be used when what you really want is arithmetic comparison (see Section 5.2 [Arithmetic], page 14) or unification.

The predicates make reference to a *standard total ordering* of terms, which is as follows:

- Variables, in a standard order (*not* related to the names of variables).
- Floats, in numeric order (e.g. -1.0 is put before 1.0).
- Integers, in numeric order (e.g. -1 is put before 1).
- Atoms, in alphabetical (i.e. character code) order.
- Compound terms, ordered first by arity, then by the name of the principal functor, then by the arguments (in left-to-right order). Recall that lists are equivalent to compound terms with principal functor `./2`.

These are the basic predicates for comparison of arbitrary terms:

Term1 `==` *Term2*

Tests if the terms currently instantiating *Term1* and *Term2* are literally identical (in particular, variables in equivalent positions in the two terms must be identical). For example, the query

```
%>dyalog -s "?-X==Y. "
```

fails (answers '`no`') because *X* and *Y* are distinct uninstantiated variables. However, the query

```
%>dyalog -s "?-X=Y,X==Y. "
Answer : Y = X
```

succeeds because the first goal unifies the two variables (see Section 5.14 [Miscellaneous], page 20).

Term1 `\==` *Term2*

Tests if the terms currently instantiating *Term1* and *Term2* are not literally identical.

Term1 `@<` *Term2*

Term *Term1* is before term *Term2* in the standard order.

Term1 `@>` *Term2*

Term *Term1* is after term *Term2* in the standard order.

Term1 `@=<` *Term2*

Term *Term1* is not after term *Term2* in the standard order.

Term1 `@>=` *Term2*
: Term *Term1* is not before term *Term2* in the standard order.

5.4 Control

`+`*P* `,` `+`*Q* Prove *P* and if it succeeds, then prove *Q*.

`+`*P* `;` `+`*Q* Prove *P* or *Q*.

`\+` `+`*Guard*
: If the guard *Guard* has a solution, fail, otherwise succeed.

`+`*Guard* `->` `+`*Q* `;` `+`*R*
: `+`*Q* is called for every possible solutions of *Guard*. `+`*R* is only called if *Guard* has no solutions.

`+`*Guard* `->` `+`*Q*
: Analogous to `+`*Guard* `->` `+`*Q*`;fail`

`true` Always succeed.

`fail` Always fail.

`wait(+`*Goal*`)`
: Wait the full completion of the evaluation of *Goal* before evaluating its continuation. Still very experimental!

The library '`call`' provide the additionnal predicate:

`call(`*Goal*`)`
: If *Goal* is instantiated to a term which would be acceptable as the body of a clause, then the goal `call(`*Term*`)` is executed exactly as if that term appeared textually in its place. There are some restrictions on *Goal*.

5.5 Error and Exception Handling

DyALog treats very poorly errors. There is only one predicate to raise errors (but no way to catch them).

`error(`*Error*`)`
: Display *Error* and fail. There is no exit of the program.

5.6 Information about the State of the Program

None.

5.7 Meta-Logic

The predicates in this section are meta-logical and perform operations that require reasoning about the current instantiation of terms or decomposing terms into their constituents. Such operations cannot be expressed using predicate definitions with a finite number of clauses.

`var(`*?X*`)` Tests whether *X* is a variable

`nonvar(`*?X*`)`
: Tests whether *X* is not a variable. This is the opposite of `var/1`.

`ground(`*?X*`)`
: Tests whether *X* is free of unbound variables.

`atom(`*?X*`)` Checks that *X* is an atom.

`integer(`*?X*`)`
: Checks that *X* is an integer.

`number(`*?X*`)`
: Checks that *X* is a number.

`atomic(`*?X*`)`
: Checks that *X* is an atom or number.

`simple(`*?X*`)`
: Checks that *X* is a variable, an atom or a number.

`compound(`*?X*`)`
: Checks that *X* is currently a term of arity `>` 0 i.e. a list or a structure.

`functor(+`*Term*`,`*?Name*`,`*?Arity*`)`
`functor(`*?Term*`,+`*Name*`,+`*Arity*`)`
: The principal functor of term *Term* has name *Name* and arity *Arity*, where *Name* is either an atom or, provided *Arity* is 0, an integer. Initially, either *Term* must be instantiated, or *Name* and *Arity* must be instantiated to, respectively, either an atom and an integer in [0..256) or an atomic term and 0. In the case where *Term* is initially uninstantiated, the result of the call is to instantiate *Term* to the most general term having the principal functor indicated.

`arg(+`*ArgNo*`,+`*Term*`,`*?Arg*`)`
: Initially, *ArgNo* must be instantiated to a positive integer and *Term* to a compound term. The result of the call is to unify *Arg* with the argument *ArgNo* of term *Term*. (The arguments are numbered from 1 upwards.)

+*Term* `=..` *?List*
?Term `=..` +*List*
: *List* is a list whose head is the atom corresponding to the principal functor of *Term*, and whose tail is a list of the arguments of *Term*. e.g.

```
%>dyalog -s "?-product(0, n, n-1) =.. L. "
Answer : L = [product,0,n,n - 1]
%>dyalog -s "?-n-1 =.. L. "
Answer : L = [-,n,1]
```

```
%>dyalog -s "?-product =.. L. "
Answer : L = [product]
```

If *Term* is uninstantiated, then *List* must be instantiated either to a list of determinate length whose head is an atom, or to a list of length 1 whose head is a number.

`name(+`*Const*`,`*?CharList*`)`
`name(`*?Const*`,+`*CharList*`)`

If *Const* is an atom or number then *CharList* is a list of the character codes of the characters comprising the name of *Const*. e.g.

```
%>dyalog -s "?-name(product,L). "
Answer : L = [0'p,0'r,0'o,0'd,0'u,0'c,0't]

%>dyalog -s "?-name(1976,L). "
Answer : L = [0'1,0'9,0'7,0'6]
```

If *Const* is uninstantiated, *CharList* must be instantiated to a list of characters. If *CharList* can be interpreted as a number, *Const* is unified with that number, otherwise with the atom whose name is *CharList*.

`atom_chars(+`*Const*`,`*?CharList*`)`
`atom_chars(`*?Const*`,+`*CharList*`)`

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be an atom.

`number_chars(+`*Const*`,`*?CharList*`)`
`number_chars(`*?Const*`,+`*CharList*`)`

The same as `name(`*Const*`,`*CharList*`)`, but *Const* is constrained to be a number.

`term_subsumer(+`*Term1*`, +`*Term2*`, -`*General*`)`

Binds *General* to the most specific term that generalizes *Term1* and *Term2*. This process is sometimes called *anti-unification*, as it is the dual of unification.

```
%>dyalog -s "?- term_subsumer(f(g(1,h(_))), f(g(_,h(1))), T). "
Answer : T = f(g(B__2,h(A__2)))

%>dyalog -s "?- term_subsumer(f(1+2,2+1), f(3+4,4+3), T). "
Answer : T = f(B__2 + C__2,C__2 + B__2)
```

5.8 Modification of the Program

None.

5.9 Internal Database

The predicates described in this section store arbitrary terms in the database without interfering with the clauses which make up the program.

`record(+`*Term*`)`

An entry associated with *Term* is added to the internal database.

`recorded(+`*Term*`)`

The internal database is searched for terms unifiable with *Term*.

`erase(+`*Term*`)`
Any entry in the internal database unifiable with *Term* is erased.

5.10 All Solutions

The predicates described in this section works on the whole set of solutions that may be computed for a goal.

`bestof(`*X*`,`*Generator*`,`*Y*`^`*Test*`)`
Test must denote a total binary relation defined as a guard and is used to compute the best *X* element for this relation in those generated by *Generator*. *X* is unified with this best element.

```
%>dyalog -s "?-bestof(X,domain(X,[1,-2,3]),Y^(X<Y)). "
Answer : X = -2
```

`iterate(` *Iterator*`,` *Generator* `)`
Iterator must be either an elementary iterator or a list of elementary iterator. An elementary iterator *New*`^(`*Init*`,`*X*`^`*Old*`^`*Updater*`)` computes the iterate value of *Init* by repeated application of *Updater* to each value *X* generated by *Generator*.

```
%>dyalog -s "?-iterate(Y^(Y is 0,X^Old^(Y is X+Old)),domain(X,[1,2,3])). "
Answer : Y = 6
```

Note that iterate doesn't fail if there is no answer *Generator* but binds *New* variables to *Init* values.

`group_by(`*Generator*`,`*Grouping*`,`*Collector*`)`
Collector of the form *New*`^`*Current*`^(`*Old*`^`*Updater*`,`*Init*`)` almagates values *Current* build by *Generator*

Note that group_by fails if *Generator* has no answer.

5.11 Debugging

None

5.12 Execution Profiling

None

5.13 Definite Clause Grammars

Definite Clause Grammars are available in DyALog using the standard notations,

Terminals to be recognized may be provided either by a PROLOG list or a set of *tokens*. A token has the form `'C'(`*Left*`,`*T*`,`*Right*`)` and means that a terminal *L* is present between the markers *Left* and *Right*. Anything may be used as markers, may integers are usually employed.

`phrase(`*:Phrase*`,`*?List*`,`*?Remainder*`)`

`phrase(`*:Phrase*`,`*?Left*`,`*?Right*`)`
: According to the current grammar rules, *Phrase* is found between *List* and *Remainder* for the first form and the markers *Left* and *Right* for the second form.

5.14 Miscellaneous

?X `=` *?Y* Defined as if by the clause `Z=Z.`; i.e. *X* and *Y* are unified.

`length(`*?List*`,`*?Length*`)`
: If *List* is instantiated to a list of determinate length, then *Length* will be unified with this length.

  If *List* is of indeterminate length and *Length* is instantiated to an integer, then *List* will be unified with a list of length *Length*. The list elements are unique variables.

  If *Length* is unbound then *Length* will be unified with all possible lengths of *List*.

`copy_term(`*?Term*`,`*?CopyOfTerm*`)`
: *CopyOfTerm* is a renaming of *Term*, such that brand new variables have been substituted for all variables in *Term*.

`argv(`*?Args*`)`
: *Args* is unified with a list of atoms of the program arguments supplied after the '`-a`' option on the command line.

`cd` Change the current working directory to the home directory.

`shell(+`*Command*`,-`*Status*`)`
: Pass *Command* to a new UNIX shell named in the Unix environment variable `$SHELL` for execution. Unify *Status* with the returned status of *Command*.

`system(+`*Command*`,-`*Status*`)`
: Pass *Command* to a new UNIX `sh` process for execution. Unify *Status* with the returned status of *Command*.

`mktemp(+`*Template*`,-`*FileName*`)`
: Interface to the C-function `mktemp(3)`. A unique file name is created and unified with *FileName*. *Template* should contain a file name with six trailing `X`s. The file name is that template with the six `X`s replaced with a letter and the process id.

`access(+`*Path*`,+`*Mode*`)`
: Tests if *Mode* is the accessability of *Path* as in the C-function `access(2)`.

`getwd(`*?Path*`)`
: Unify *Path* with the atom representation of the current working directory.

`getenv(+`*Name*`,`*?Value*`)`
: Unify *Value* with the atom representation of the value of the environment variable given by *Name*.

`gensym`(-*Id*)

Unify *Id* with a fresh integer.

`domain`(*x?X*,+*List*)

A built-in oriented version of `member/2`.

6 Directives

Directives are introduced by `:-/1` and are used to extend the compiler or the reader/printer.

Unary directives such as `require/1` are prefix operators with high precedence, allowing to write for instance:

```
:-require 'foo.pl','bar.pl'.
```

6.1 General Directives

General directives are used to govern the parser and reader.

`op(`+*Prec*`,`+*Kind*`,`+*Op_List*`)`
: Declare all symbols in *Op_List* as operators of precedence *Prec* and nature *Kind*.

`hilog` +*Symbol_List*
: Declare all symbols in *Symbol_List* as hilog symbols.

`features(`+*Symbol_List*`,`+*Feature_list*`)`
: Declare all symbols in *Symbol_List* as feature functor with associated feature list *Feature_list*. Element of *Feature_list* should be symbols.

6.2 Compiler Directives

`include` +*Filename_List*

`require` +*Filename_List*

`resource` +*Filename_List*

`mode(`+*Pred_Spec*`,`+*Mode_Pattern*`)`

`dcg_mode(`+*Pred_Spec*`,`+*Mode_Pattern*`,`+*Left_Mode_Pattern*`,`+*Right_Mode_Pattern*`)`

`extensional` +*Pred_Spec*

`prolog` +*Pred_Spec*

`rec_prolog` +*Pred_Spec*

`lco` +*Pred_Spec*

`parse_mode` *Mode*

`cmode(`*Mode*`)`

`xcompiler(`*Clause*`)`

`bmg_stacks` +*Symbol_List*

`bmg_island(`+*Island*`,`+*Island_Stacks*`)`

`bmg_pushable(`+*Pred_Spec*`,`+*Stack_List*`)`

6.3 Directive Files

7 Typed Feature Structures

Typed Feature Structures a la Carpenter are available in DyALog (in an experiemental way). They extend standard Feature Structures by considering that (1) feature structure functors are types in some type hierarchy, (2) features are inherited from type to subtype and (3) feature values must satisfy type constraints.

For instance, '`list.def`' specify a small hierarchy for TFS corresponding to lists.

```
bot sub [atom,list].
    atom sub [].
    list sub [e_list,ne_list].
        e_list sub [].
        ne_list sub [] intro [hd:atom,tl:list].
```

This description says that `bot` is the most general type with subtypes `atom` and `list`. Similarly, `list` has two subtypes `e_list` (for *empty list*) and `ne_list` (for *non empty list*). `ne_list` introduces two new features, namely `hd` and `tl` whose most general type should be respectively `atom` and `list`.

The set of features associated with `type` is given by all features introduced by `type` and its super types. A feature may be introduced again by `type` with a more specific type.

Any tfs built on `type` may be instanciated to a new tfs built on a subtype of `type`. For instance, `list{}` generalizes both `e_list{}` and `ne_list{hd=>atom{},tl=>list{}}`.

Subsumption checking as well as unification have therefore to be extended to handle *type shifting*. This is done by linking executable with a C library generated from the description file.

Actually, the generated C library need also to be dl-opened by the compiler to extend program reading and performs some immediate unifications.

Let suppose we want to use the following implementation of append:

```
%> cat tfs_append.pl

append(e_list{}, Y::list{}, Y).

append( A::tl=>X,Y::list{}, B::tl=>Z) :-
        A .> hd .= B.> hd,
        append(X,Y,Z).

?-X=tl=>tl=>e_list{},Y=tl=>_,append(X,Y,Z).
```

In file '`tfs_append.pl`', `Y::list{}` denotes *immediate unification* performed by the compiler between `Y` and `list{}`. Notation `tl=>X` introduces the most general tfs with feature `tl` bound to `X`, i.e. `ne_list{hd=>atom{},tl=>X::list{}}`. Expression `A .> hd .= B .> hd` is an alternate way to specify the unifiability of values given as feature pathes: here, we unify value of feature `hd` of tfs `A` with value of feature `hd` of tfs `B`.

To compile '`tfs_append.pl`', we first need to generate the C library from '`list.def`'.

```
%> tfs2lib list.def
```

The resulting library is called '`liblist.so.0`'. It will used by the compiler to be able to correctly read '`tfs_append.pl`' and linked with the executable to extend unification and subsumption.

```
%> dyacc -tfs list tfs_append.pl -o tfs_append
```

Both the compiler (`dyalog`) and the executable (`tfs_append`) should be able to locate library '`liblist.so.0`', either by moving the library in some known library directory or by setting, for instance, the environment variable *LD_LIBRARY_PATH*.

An alternate way is to use the option -libtool <libtoo_pgm> for both `tfs2lib` and `dyacc`.

```
%> tfs2lib list.def -libtool libtool
%> dyacc -tfs list -libtool libtool tfs_append.pl -o tfs_append
```

The library is now called '`liblist.la`' and is actually a shell wrapper to the true library.

8 Tree Adjoining Grammars

DyAlog can compile Feature Tree Adjoining Grammars [FTAGs]. This extension is based on `ftp://ftp.inria.fr/INRIA/Projects/Atoll/Eric.Clergerie/SD2SA.ps.gz`. TAGs are compiled into (meta) transitions of a 2-stack automaton, which are then compiled into DyALog objects and application functions for a run-time tabular evaluation.

For instance, the following grammar defines the langage `a^nb^nec^nd^n`

```
tree s("e").
auxtree -s("a",s("b",*s,"c"),"d").
```

The first tree of this program is an initial tree reading the terminal `"e"` and allowing an adjonction on the node s.

The second tree is an auxiliary tree. Its foot node is the one marked by `*` and adjonction is not allowd on the node marked by `-`.

Mandatory adjonction may be marked by prefixing a node by `++`.

TAG non terminals may be called from a Prolog program using predicate `tag_phrase/3`.

Nodes may be named and decorated with a pair of `top` and `bot` attributes.

Following the XTAG architecture, trees may be named and grouped in set of families. Each tree may have an anchor node, a distinguished node marked `<>`. Tree are related to lemma

The following is a fragment from a small french FTAG for verb `donne` (gives).

```
%% Specify wich parameter of trees in tn1pn2 may be instanciated
%% with information from the lexicon
:-tag_anchor{ name => tn1pn2,
              coanchors => [ p_2 ],
              equations => [ [A]^(top=np{ restr => A } at np_2),
                             [B]^(top=np{ restr => B } at np_0)
                           ]
            }.

%% Define tree tn1pn2 in family tn1pn2
%% anchored on a verb
tag_tree{ name => tn1pn2,
          family => tn1pn2,
          tree=> tree
        bot=s{mode => X2, inv => (-)}
        at s(
             id=np_0
            and top=np{num => X0, pers => X1, wh => (-)}
            at np,
             top=vp{num => X0, pers => X1, mode => X2}
            and bot=vp{mode => X3, num => X6, pers => X7}
            at vp(
                  bot=v{mode => X3, num => X6, pers => X7}
                 at <> v,
                  np,
                  pp(
```

```
                          id=p_2 at p,
                          id=np_2 at np
                         )
                      )
                 )
             }.

    %% Define lemma entry in the lexicon for verb DONNER (to give)
    tag_lemma('*DONNER*',v,
              tag_anchor{ name=>tn1pn2,
                           coanchors=>[p_2=],
                           equations=>[top = np{ restr=>plushum } at np_0,
                                       top = np{ restr=>plushum } at np_2]}
             ).

    %% Define morph entry in the lexicon for verb donne (gives)
    tag_lexicon(donne, '*DONNER*', v,
                v{ mode => mode[ind, subj], num => sing }).
```

Modulation may be applied on TAG non terminals.

9 Range Concatenation Grammars

Range Concatenation Grammars is a formalism introduced by Pierre Boullier. They provide an elegant way to specify non-contiguous or even overlapping constituant by expressing constraints on sub-ranges of the input string.

For instance, the following grammar defines the language `a^nb^nc^nd^n`:

```
s (X@Y@Z) --> a (X,Y,Z).
a ("a"@X,"b"@Y,"c"@Z) --> a (X,Y,Z).
a ("","","") --> true.
```

Note that the range arguments are separated from their predicate by a whitespace (Hilog notation).

RCG non terminals may be called from a logic program using `rcg_phrase/1`, for instance `rcg_phrase(s (0:N)`.

RCG should be compiled with option `-rcg` to distinguish them from DCG.

RCG may be compiled with or without option `-parse`, depending if the grammar is to be used to parse from PROLOG lists or token databases.

RCG non terminals may be decorated with attributes and `{}` may be used to escape to PROLOG. For instance, the previous program may be rewritten to count.

```
s(N) (X@Y@Z) --> a(N) (X,Y,Z).
a(N) ("a"@X,"b"@Y,"c"@Z) --> a(M) (X,Y,Z), N is M+1.
a(0) ("","","") --> true.
```

A RCG predicate is characterized by its Prolog arity and its range arity. For instance, non-terminal `a(N) (X,Y,Z)` in the previous program formelly corresponds to predicate `rcg(a/1,3)`.

Directives `prolog/1` or `rec_prolog/1` apply on RCG predicates to change their tabulation status.

Directive `mode/2` also applies to alter their modulation status. The modulation only acts on the PROLOG arguments.

For instance, to be bottom up on the counting argument, use

```
:-mode([rcg(s/1,1),rcg(a/1.3)],+(-)).
```

Appendix A Standard Operators

```
:-op( 1200, xfx, [(:-),(-->)] ).
:-op( 1200,  fx, [(:-),(?-)] ).
:-op( 1100, xfy, [(;)] ).
:-op( 1050, xfy, [->] ).
:-op( 1000, xfy, [',']  ).
:-op(  900,  fy, [\+,spy,nospy] ).
:-op(  700, xfx, [=,is,=..,==,@<,@>,@=<,@>=,\==,=:=,=\=,<,>,=<,>=] ).
:-op(  600, xfy, [:] ).
:-op(  500, yfx, [+,-,\\/,/\\] ).
:-op(  500,  fx, [-,+] ).
:-op(  400, yfx, [*,/,//,<<,>>,div] ).
:-op(  300, xfx, [mod] ).
:-op(  200, xfy, [^] ).
:-op(  900, xfy, [&] ).
:-op(  700, xf , [?] ).
:-op(  700, xfx, [isagg] ).
```

Index of Built-Ins

#

/2, bitwise exclusive or ... 14

*

* /2, multiplication ... 14

,

,/2, conjunction ... 16

-

- /1, negation ... 14
- /2, subtraction ... 14
-> /2 ;/2, if then else ... 16
-> /2, if then ... 16

/

// /2, integer division ... 14
/\ /2, bitwise conjunction ... 14

;

;/2, disjunction ... 16

=

= /2, unification ... 20
=../2, univ ... 17
=:= /2, arithmetic equal ... 14
== /2, equality of terms ... 15
=\= /2, arithmetic not equal ... 15
=< /2, arithmetic less or equal ... 15

@

@=< /2, term less or equal ... 15
@> /2, term greater than ... 15
@>= /2, term greater or equal ... 16
@< /2, term less than ... 15

+

+ /1, identity ... 14
+ /2, addition ... 14

>

> /2, arithmetic greater than ... 15
>= /2, arithmetic greater or equal ... 15
>> /2, right shift ... 14

\

\ /1, bitwise negation ... 14
\/ /2, bitwise disjunction ... 14
\== /2, inequality of terms ... 15
\+ /1, not provable ... 16

<

< /2, arithmetic less than ... 15
<< /2, left shift ... 14

A

abs/1, absolute value ... 14
absolute_file_name/2 ... 12
access/2 ... 20
add_stream_alias/2 ... 13
arg/3 ... 17
argv/1 ... 20
atom/1 ... 17
atom_chars/2 ... 18
atomic/1 ... 17

B

bestof/3 ... 19
bmg_island/2 (directive) ... 22
bmg_pushable/2 (directive) ... 22
bmg_stacks/1 (directive) ... 22

C

call/1 ... 16
cd/0 ... 20
close/1 ... 12
cmode/1 (directive) ... 22
compound/1 ... 17
copy_term/2 ... 20
current_input/1 ... 12
current_output/1 ... 12

D

dcg_mode/4 (directive) ... 22
display/1 ... 12
domain/2 ... 21

E

erase/1 ... 19
error/1 ... 16

F

fail/0 ... 16
features/2 (directive) ... 22
flush_output/0 ... 13
flush_output/1 ... 13
from_alias_to_stream/2 ... 13
functor/3 ... 17

G

gensym/1 ... 21
getwd/1 ... 20
ground/1 ... 17
group_by/3 ... 19

H

hilog/1 (directive) ... 22

I

include/1 (directive) ... 22
integer/1 ... 17
is/2 ... 14
iterate/2 ... 19

L

lco/1 (directive) ... 22
length/2 ... 20

M

max/2, maximum value ... 14
min/2, minimum value ... 14
mktemp/2 ... 20
mod/2, integer remainder ... 14
mode/2 (directive) ... 22

N

name/2 ... 18
nonvar/1 ... 17
number/1 ... 17
number_chars/2 ... 18

O

op/3 (directive) ... 22
open/3 ... 12

P

parse_mode/1 (directive) ... 22
phrase/3 ... 20
prolog/1 (directive) ... 22

R

read/1 ... 11
read/2 ... 11
read_term/2 ... 11
read_term/3 ... 11
rec_prolog/1 (directive) ... 22
recorded/1 ... 18
require/1 (directive) ... 22
resource/1 (directive) ... 22

S

see/1 (dec10 library) ... 13
seeing/1 (dec10 library) ... 13
seen/0 (dec10 library) ... 13
set_input/1 ... 13
set_output/1 ... 13
shell/2 (unix/1 option) ... 20
simple/1 ... 17
system/2 ... 20

T

tell/1 (dec10 library) ... 13
telling/1 (dec10 library) ... 13
term_subsumer/3 ... 18
told/0 (dec10 library) ... 13
true/0 ... 16

V

var/1 ... 17

W

wait/1 ... 16
write/1 ... 11
write/2 ... 11
writeq/1 ... 12
writeq/2 ... 12

X

xcompiler/1 (directive) ... 22

Concept Index

A

arithmetic 14

C

comparing terms 15

E

end of file 2
EOF 2

K

keyboard 2

M

meta-logical predicate 17
mode spec 2

N

notation 2

P

predicate spec 2

S

standard order 15

T

term comparison 15

Table of Contents

Introduction 1

Notational conventions 2

1 Installation 3

1.1 Obtaining DyALog 3
1.2 Supported Machines 3
1.3 Installation 3
1.3.1 Configuring DyALog 3
1.3.2 Installing DyALog 3

2 Using DyALog 4

2.1 Dyacc 4
2.2 Dyalog 5
2.3 Dyam2asm 5
2.4 DyALog executables 6
2.5 An example 6

3 Behind the screen 7

3.1 LPDA 7
3.2 Dynamic Programming 7
3.3 Compilation Process 7
3.3.1 From programs to LPDA 7
3.3.2 From LPDA to Abstract Machine Code 7
3.3.3 Emitting C code 7
3.4 Execution 7

4 Syntax 8

4.1 Terms 8
4.1.1 Standard terms 8
4.1.2 Immediate unification 8
4.1.3 Operators 8
4.1.4 Feature Terms 8
4.1.5 Enumeration variables 9
4.1.6 Hilog terms 9
4.2 Programs 9
4.2.1 Clauses 9
4.2.2 Definite Clause Grammars 10
4.2.3 Directives 10

5 Built-In Predicates 11
5.1 Input / Output 11
5.1.1 Reading-in Programs 11
5.1.2 Input and Output of Terms 11
5.1.3 Character Input/Output 12
5.1.4 Stream IO 12
5.1.5 Socket IO 13
5.1.6 DEC-10 Prolog File IO 13
5.2 Arithmetic 14
5.3 Comparison of Terms 15
5.4 Control 16
5.5 Error and Exception Handling 16
5.6 Information about the State of the Program 16
5.7 Meta-Logic 17
5.8 Modification of the Program 18
5.9 Internal Database 18
5.10 All Solutions 19
5.11 Debugging 19
5.12 Execution Profiling 19
5.13 Definite Clause Grammars 19
5.14 Miscellaneous 20

6 Directives 22
6.1 General Directives 22
6.2 Compiler Directives 22
6.3 Directive Files 22

7 Typed Feature Structures 23

8 Tree Adjoining Grammars 25

9 Range Concatenation Grammars 27

Appendix A Standard Operators 28

Index of Built-Ins 29

Concept Index 32