Logic Programming

Efficient Prolog. I/O

November 18, 2015

The procedural aspect of Prolog

- While Prolog is described as a declarative language, one can see Prolog clauses from a procedural point of view:

```
in(X, usa):-
        in(X, mississippi).
```

  The above can be seen:
  - from a declarative point of view: “X is in the USA if X is in Mississippi”,
  - from a procedural point of view: “To prove that X is in the USA, prove X is in Mississippi”, or “To find X in USA, (it is sufficient to) find them in Mississippi”.
- Procedural programming languages can also contain declarative aspects. Something like

```
x = y + z;
```

  can be read
  - declaratively, as the equation $x = y + z$,
  - procedurally: load y, load z, store x.

The need to understand the procedural/declarative aspects

- The declarative/procedural aspects are not "symmetrical": there are situations where one not understanding one aspect can lead to problems.
- For procedural programs: A = (B + C) + D and A = B + (C +D) appear to have equivalent declarative readings but:
  - imagine the biggest number that can be represented is 1000,
  - then for B = 501, C = 501, D = -3, the two expressions yield totally different results!
- The same can happen in Prolog. Declaratively, the following is correct:

```
ancestor(A, C):-
        ancestor(A, B),
        ancestor(B, C).
```

However, ignoring its procedural meaning, this can lead to infinite loops (when B and C are both unknown).

- The task of a Prolog programmer is to build a model of the problem and to represent it in Prolog.
- Knowledge about this model can improve performance significantly.

  ?−horse(X), gray(X). will find the answer much faster than ?−gray(X), horse(X). in a model with 1000 gray objects and 10 horses.
- Narrowing the search can be even more subtle:

```
set_equivalent(L1, L2):-
        permute(L1, L2).
```

  i.e. to find whether two lists are set-equivalent it is enough to see whether they are permutations of eachother. But for N element lists, there are $N!$ permutations (e.g. for 20 elements, 2.4×10^{18} possible permutations).

- Now considering a faster program:

```
set_equivalent(L1, L2):-
        sort(L1, L3),
        sort(L2, L3).
```

i.e. two lists are set equivalent if their sorted versions are the same. And sorting can be done in $NlogN$ steps (e.g. approx 86 steps for 20 element lists).

- When patterns are involved, unification can do some of the work that the programmer may have to do.
- E.g. consider variants the predicate that detects lists with 3 elements:
  - ```
    has_3_elements(X):-
                    length(X, N),
                    N = 3.
    ```
  - ```
    has_3_elements([_, _, _]).
    ```
- Also consider the predicate for swapping the first two elements from a list:

  ```
  swap_first_2([A, B| Rest], [B, A| Rest]).
  ```

  Letting unification work saves having to go through the whole list.

- Atoms are represented in Prolog in a symbol table where each atom appears once - a process called *tokenization*.

- Atoms are represented in Prolog in a symbol table where each atom appears once - a process called *tokenization*.
- Atoms in a program are replaced by their address in the symbol table.

- Atoms are represented in Prolog in a symbol table where each atom appears once - a process called *tokenization*.
- Atoms in a program are replaced by their address in the symbol table.
- Because of this:

```
f('What a long atom this appears to be',
  'What a long atom this appears to be',
  'What a long atom this appears to be').
```

will actually take less memory than g(a, b, c)

- Atoms are represented in Prolog in a symbol table where each atom appears once - a process called *tokenization*.
- Atoms in a program are replaced by their address in the symbol table.
- Because of this:

  ```
  f('What a long atom this appears to be',
    'What a long atom this appears to be',
    'What a long atom this appears to be').
  ```

  will actually take less memory than `g(a, b, c)`
- Comparison of atoms can be performed very fast because of tokenization.

- Atoms are represented in Prolog in a symbol table where each atom appears once - a process called *tokenization*.
- Atoms in a program are replaced by their address in the symbol table.
- Because of this:

```
f('What a long atom this appears to be',
 'What a long atom this appears to be',
 'What a long atom this appears to be').
```

  will actually take less memory than `g(a, b, c)`
- Comparison of atoms can be performed very fast because of tokenization.
- For example a \= b and aaaaaaaaa \= aaaaaaaab can both be done in the same time, without having to "parse" the whole atom names.

Continuations, backtracking points

- Consider the following:

```
a :- b, c.
a :- d.
```

Continuations, backtracking points

- Consider the following:

```
a :-  b ,  c .
a :-  d .
```

- For ?- a., when b is called, Prolog has to save in the memory:

Continuations, backtracking points

- Consider the following:

```
a :- b, c.
a :- d.
```

- For ?– a., when b is called, Prolog has to save in the memory:
  - *the continuation*, i.e. what has to be done after returning with success from b (i.e. c),

Continuations, backtracking points

- Consider the following:

```
a :-  b ,  c .
a :-  d .
```

- For ?- a., when b is called, Prolog has to save in the memory:
  - *the continuation*, i.e. what has to be done after returning with success from b (i.e. c),
  - *the backtrack point*, i.e. where can an alternative be tried in case of returning with failure from b (i.e. d).

Continuations, backtracking points

- Consider the following:

```
a :- b, c.
a :- d.
```

- For ?– a., when b is called, Prolog has to save in the memory:
  - *the continuation*, i.e. what has to be done after returning with success from b (i.e. c),
  - *the backtrack point*, i.e. where can an alternative be tried in case of returning with failure from b (i.e. d).
- For recursive procedures the continuation and backtracking point have to be remembered for each of the recursive calls.

Continuations, backtracking points

- Consider the following:

```
a :-  b ,  c .
a :-  d .
```

- For ?- a., when b is called, Prolog has to save in the memory:
  - *the continuation*, i.e. what has to be done after returning with success from b (i.e. c),
  - *the backtrack point*, i.e. where can an alternative be tried in case of returning with failure from b (i.e. d).
- For recursive procedures the continuation and backtracking point have to be remembered for each of the recursive calls.
- This may lead to large memory requirements

Tail recursion

- If a recursive predicate has no continuation, and no backtracking point, Prolog can recognize this and will not allocate memory.

Tail recursion

- If a recursive predicate has no continuation, and no backtracking point, Prolog can recognize this and will not allocate memory.
- Such recursive predicates are called *tail recursive* (the recursive call is the last in the clause and there are no alternatives).

Tail recursion

- If a recursive predicate has no continuation, and no backtracking point, Prolog can recognize this and will not allocate memory.
- Such recursive predicates are called *tail recursive* (the recursive call is the last in the clause and there are no alternatives).
- They are much more efficient than the non-tail recursive variants.

Tail recursion

- If a recursive predicate has no continuation, and no backtracking point, Prolog can recognize this and will not allocate memory.
- Such recursive predicates are called *tail recursive* (the recursive call is the last in the clause and there are no alternatives).
- They are much more efficient than the non-tail recursive variants.
- The following is tail recursive:

```
test1(N):- write(N),nl,NewN is N+1,test1(NewN).
```

In the above write writes (prints) the argument on the console and succeeds, nl moves on a new line and succeeds. The predicate will print natural numbers on the console until the resources run out (memory or number representations limit).

- The following is not tail recursive (it has a continuation):

```
test2(N):-  write(N),nl,NewN is N+1,
            test2(NewN),nl.
```

When running this, it will run out of memory relatively soon.

- The following is not tail recursive (it has a backtracking point):

```
test3(N):-  write(N),nl,NewN is N+1,
            test3(NewN).
test3(N):-  N<0.
```

- The following is tail recursive (the alternative clause comes before the recursive clause so there is no backtracking point for the recursive call):

```
test3a(N):-  N<0.
test3a(N):-  write(N), nl, NewN is N+1,
             test3a(NewN).
```

- The following is not tail recursive (it has alternatives for predicates in the recursive clause preceding the recursive call, so backtracking may be necessary):

```
test4(N):- write(N), nl, m(N, NewN),
           test4(NewN).
m(N, NewN):- N >= 0, NewN is N + 1.
m(N, NewN):- N < 0, NewN is (-1)*N.
```

Making recursive predicates tail recursive

- If a predicate is not tail recursive because it has backtracking points, then it can be made so by using the cut before the recursive call.

Making recursive predicates tail recursive

- If a predicate is not tail recursive because it has backtracking points, then it can be made so by using the cut before the recursive call.
- The following are now tail recursive:

```
test5(N):-  write(N),nl,NewN is N+1,!,
            test5(NewN).
test5(N):- N<0.
test6(N):-  write(N), nl, m(N, NewN), !,
            test6(NewN).
m(N, NewN):- N >= 0, NewN is N + 1.
m(N, NewN):- N < 0, NewN is (-1)*N.
```

- Note that tail recursion can be indirect. The following is tail recursive:

```
test7(N):- write(N), nl, test7a(N).
test7a(N):- NewN is N+1, test7(NewN).
```

In the above we have mutual recursion, but note that test7a is just used to rename part of the test7 predicate.

Summary: tail recursion

- In Prolog, tail recursion exists when:
  - the recursive call is the last subgoal in the clause,
  - there are no untried alternative clauses,
  - there are no untried alternatives for any subgoal preceding the recursive call in the same clause.

- Consider the program:

```
a(b).
a(c).

d(e).
d(f).
```

and the query ?−d(f).

- Consider the program:

```
a(b).
a(c).

d(e).
d(f).
```

  and the query ?−d(f).

- Contrary to the expectation, most Prolog implementations will not have to go through all the knowledge base.

- Consider the program:

```
a(b).
a(c).

d(e).
d(f).
```

  and the query ?−d(f).
- Contrary to the expectation, most Prolog implementations will not have to go through all the knowledge base.
- Prolog uses *indexing* over the functor name and the first argument.

- Consider the program:

```
a(b).
a(c).

d(e).
d(f).
```

  and the query ?–d(f).
- Contrary to the expectation, most Prolog implementations will not have to go through all the knowledge base.
- Prolog uses *indexing* over the functor name and the first argument.
- These indices will be stored as a hash table or something similar for fast access.

- Consider the program:

```
a(b).
a(c).

d(e).
d(f).
```

  and the query ?−d(f).

- Contrary to the expectation, most Prolog implementations will not have to go through all the knowledge base.
- Prolog uses *indexing* over the functor name and the first argument.
- These indices will be stored as a hash table or something similar for fast access.
- Therefore, Prolog will find d(f) directly.

- Using indexing can make predicates be tail recursive when they would not be:

```
test8(0):-write('Still going'), nl, test8(0).
test8(-1).
```

The second clause is not an alternative to the first because of indexing.

- Using indexing can make predicates be tail recursive when they would not be:

```
test8(0):-write('Still going'), nl, test8(0).
test8(-1).
```

  The second clause is not an alternative to the first because of indexing.

- Note, however, that indexing works only when the first argument of the predicate is instantiated.

- Consider some built-in predicates in Prolog, as presented in the help section of the program:

```
append(?List1, ?List2, ?List3)
```

Succeeds when List3 unifies with the concatenation of List1 and List2. The predicate can be used with any instantiation pattern (even three variables).

```
-Number is +Expr        [ISO]
```

True if Number has successfully been unified with the number Expr evaluates to. If Expr evaluates to a float that can be represented using an integer (i.e, the value is integer and within the range that can be described by Prolog's integer representation), Expr is unified with the integer value.

- The above examples use a notation (documentation) convention in Prolog: when describing the predicate, use *mode indicators* for its arguments:

Note that the above description does not ensure guarantee what would happen if the argument is used in another mode. For that matter, it does not even guarantee the intended behavior.

- The above examples use a notation (documentation) convention in Prolog: when describing the predicate, use *mode indicators* for its arguments:
  - $+$ describes an argument that should already be instantiated when the predicate is called,

Note that the above description does not ensure guarantee what would happen if the argument is used in another mode. For that matter, it does not even guarantee the intended behavior.

- The above examples use a notation (documentation) convention in Prolog: when describing the predicate, use *mode indicators* for its arguments:
  - \+ describes an argument that should already be instantiated when the predicate is called,
  - \- denotes an argument that is normally not instantiated until this predicate instantiates it,

Note that the above description does not ensure guarantee what would happen if the argument is used in another mode. For that matter, it does not even guarantee the intended behavior.

- The above examples use a notation (documentation) convention in Prolog: when describing the predicate, use *mode indicators* for its arguments:
  - \+ describes an argument that should already be instantiated when the predicate is called,
  - \- denotes an argument that is normally not instantiated until this predicate instantiates it,
  - ? denotes an argument that may or may not be instantiated,

Note that the above description does not ensure guarantee what would happen if the argument is used in another mode. For that matter, it does not even guarantee the intended behavior.

- The above examples use a notation (documentation) convention in Prolog: when describing the predicate, use *mode indicators* for its arguments:
  - \+ describes an argument that should already be instantiated when the predicate is called,
  - \- denotes an argument that is normally not instantiated until this predicate instantiates it,
  - ? denotes an argument that may or may not be instantiated,
  - @ is used by some programmers to indicate that the argument contains variables that must not be instantiated.

  Note that the above description does not ensure guarantee what would happen if the argument is used in another mode. For that matter, it does not even guarantee the intended behavior.

- There are two styles of I/O in Prolog:
  - Edinburg style I/O is the legacy style, still supported by Prolog implementations. It is relatively simple to use but has some limitations.
  - ISO I/O is the standard style, supported by all Prolog implementations.
- There are some overlaps between the two styles.

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.
- The predicate nl moves the "cursor" at a new line.

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.
- The predicate nl moves the “cursor” at a new line.

```
?- write('Hello there'), nl, write('Goodbye').
   Hello there
   Goodbye
   true.
```

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.
- The predicate nl moves the "cursor" at a new line.

```
?- write('Hello there'), nl, write('Goodbye').
   Hello there
   Goodbye
   true.
```

- Note that quoted atoms are displayed without quotes. The variant writeq of write will also display the quotes.

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.
- The predicate nl moves the "cursor" at a new line.

```
?- write('Hello there'), nl, write('Goodbye').
   Hello there
   Goodbye
   true.
```

- Note that quoted atoms are displayed without quotes. The variant writeq of write will also display the quotes.

```
?- write(X).
   _G243
    true.
?- write("some str").
[115, 111, 109, 101, 32, 115, 116, 114]
true.
```

Writing terms

- The built-in predicate write takes any Prolog term and displays it on the screen.
- The predicate nl moves the “cursor” at a new line.

```
? - write('Hello there'), nl, write('Goodbye').
  Hello there
  Goodbye
  true.
```

- Note that quoted atoms are displayed without quotes. The variant writeq of write will also display the quotes.

```
?- write(X).
   _G243
    true.
?- write("some str").
[115, 111, 109, 101, 32, 115, 116, 114]
true.
```

- Note that Prolog displays the internal representation of terms. In particular, the internal representation of variables.

Reading terms

- The predicate read accepts any Prolog term from the keyboard (typed in Prolog syntax, followed by the period).

```
?- read(X).
|: hello.
X = hello.

?- read(X).
|: 'hello there'.
X = 'hello there'.

?- read(X).
|: hello there.
ERROR: Stream user_input:0:37
Syntax error: Operator expected
```

```
?- read(hello).
|: hello.
true.

?- read(hello).
|: bye.
false.


?- read(X).
|: mother(Y, ada).
X = mother(_G288, ada).
```

- The read predicate succeeds if its argument can be unified with the term given by the user (if this is a term). The examples above illustrate several possible uses and situations.

File handling

- The predicate see takes a file as argument, the effect is to open the file for reading such that Prolog gets input from that file rather than the console.

File handling

- The predicate see takes a file as argument, the effect is to open the file for reading such that Prolog gets input from that file rather than the console.
- The predicate seen closes all open files, input now comes again from the console.

File handling

- The predicate see takes a file as argument, the effect is to open the file for reading such that Prolog gets input from that file rather than the console.
- The predicate seen closes all open files, input now comes again from the console.

```
?- see('myfile.txt'),
read(X),
read(Y),
read(Z),
seen.
```

File handling

- The predicate see takes a file as argument, the effect is to open the file for reading such that Prolog gets input from that file rather than the console.
- The predicate seen closes all open files, input now comes again from the console.

```
?- see('myfile.txt'),
read(X),
read(Y),
read(Z),
seen.
```

- When a file is opened, Prolog will keep track of the position of the "cursor" in that file.

- One can switch between several open files:

```
?- see('aaaa'),
   read(X1),
   see('bbbb'),
   read(X2),
   see('cccc'),
   read(X3),
   seen.
```

- The predicate tell opens a file for writing and switches the output to it.

- The predicate tell opens a file for writing and switches the output to it.
- The predicate told closes all files opened for writing and returns the output to being the console.

```
?- tell('myfile.txt'),
   write('Hello there'),
   nl,
   told.
```

- The predicate `tell` opens a file for writing and switches the output to it.
- The predicate `told` closes all files opened for writing and returns the output to being the console.

```
?- tell('myfile.txt'),
   write('Hello there'),
   nl,
   told.
```

- Several files can be opened and written into:

```
?- tell('aaaa'),
   write('first line of aaaa'), nl,
   tell('bbbb'),
   write('first line of bbbb'), nl,
   tell('cccc'),
   write('first line of cccc'), nl,
   told.
```

Character level I/O

- The predicate put writes one character (integer representing the ASCII code corresponding to the character).

```
?- put(42).
*
true.
```

Character level I/O

- The predicate put writes one character (integer representing the ASCII code corresponding to the character).

```
?- put(42).
*
true.
```

- The predicate get reads one character from the default input (console).

```
?- get(X).
|     %

X = 37.
```

Character level I/O

- The predicate put writes one character (integer representing the ASCII code corresponding to the character).

```
?- put(42).
*
true.
```

- The predicate get reads one character from the default input (console).

```
?- get(X).
|     %

X = 37.
```

- In SWI Prolog, put can also handle nonprinting characters:

```
?- write(hello), put(8), write(bye).
hellbye
true.
```

Complete Information: SWI-Prolog Manual

- For exact details of the Edinburgh style I/O predicates in SWI Prolog, consult [Wielemaker, 2008] (also available in SWI Prolog by calling ? − help.).

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).
- ISO I/O predicates are provided for:

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).
- ISO I/O predicates are provided for:
  - Open and close streams in different modes.

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).
- ISO I/O predicates are provided for:
  - Open and close streams in different modes.
  - Inspecting the status of a stream, as well as other information.

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).
- ISO I/O predicates are provided for:
  - Open and close streams in different modes.
  - Inspecting the status of a stream, as well as other information.
  - Reading/writing is done in streams.

Streams

- ISO standard I/O in Prolog considers the notion of a stream (open files or file-like objects).
- ISO I/O predicates are provided for:
  - Open and close streams in different modes.
  - Inspecting the status of a stream, as well as other information.
  - Reading/writing is done in streams.
- There are two special streams that are always open: `user_input` and `user_output`.

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

    Filename indicates the file name (implementation, OS dependent),

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

    Filename indicates the file name (implementation, OS dependent),

    Mode is one of read, write, append,

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a "handle" for the file,

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a “handle” for the file,
  Options is a (possibly empty) list of options. Options include:

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a "handle" for the file,
  Options is a (possibly empty) list of options. Options include:

  - type(text) (default) or type(binary),

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a “handle” for the file,
  Options is a (possibly empty) list of options. Options include:
  - type(text) (default) or type(binary),
  - reposition (true) or reposition (false) (the default) – indicating whether it is possible to skip back or forward to specified positions,

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a “handle” for the file,
  Options is a (possibly empty) list of options. Options include:

  - type(text) (default) or type(binary),
  - reposition (true) or reposition (false) (the default) – indicating whether it is possible to skip back or forward to specified positions,
  - alias (Atom) – a name (atom) for the stream,

Opening streams

- The predicate open(Filename, Mode, Stream, Options) opens a stream, where:

  Filename indicates the file name (implementation, OS dependent),
  Mode is one of read, write, append,
  Stream is a "handle" for the file,
  Options is a (possibly empty) list of options. Options include:

  - type(text) (default) or type(binary),
  - reposition (true) or reposition (false) (the default) – indicating whether it is possible to skip back or forward to specified positions,
  - alias (Atom) – a name (atom) for the stream,
  - action for reading past the end of the line: eof_action (error) - raise an error condition, oef_action (eof_code) - return an error code, eof_action (reset) - to examine the file again (in case it was updated e.g. by another concurrent process).

- Example:

```
test:-
open('file.txt', read, MyStream, [type(text)]),
read_term(MyStream, Term, [quoted(true)]),
close(MyStream), write(Term).
```

Closing streams

- The predicate close(Stream, Options) closes Stream with Options.

Closing streams

- The predicate close(Stream, Options) closes Stream with Options.
- close(Stream) is the version without options.

Closing streams

- The predicate close(Stream, Options) closes Stream with Options.
- close(Stream) is the version without options.
- Options include force(false) (default) and force(true) - even if there is an error (e.g. the file was on a removable storage device which was removed), the file is considered closed, without raising an error.

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:
  - file_name (...) ,

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:
  - file_name (...) ,
  - mode(M),

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:
  - file_name (...) ,
  - mode(M),
  - alias (A),

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:
  - file_name (...) ,
  - mode(M),
  - alias (A),
  - etc, consult the documentation [Wielemaker, 2008] for the rest of the options.

Stream properties

- The predicate stream_property(Stream, Property) can be used to get properties like:
  - file_name (...) ,
  - mode(M),
  - alias (A),
  - etc, consult the documentation [Wielemaker, 2008] for the rest of the options.
- Example:

```
?- stream_property(user_input, mode(What)).
What = read.
```

Reading terms

- Predicates for reading terms:

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),

  read_term(Term, Options), using the current input stream,

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),
  read_term(Term, Options), using the current input stream,
  read(Stream, Term) like the above, without the options,

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),
  read_term(Term, Options), using the current input stream,
  read(Stream, Term) like the above, without the options,
  read(Term) like the above, from current input.

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),
  read_term(Term, Options), using the current input stream,
  read(Stream, Term) like the above, without the options,
  read(Term) like the above, from current input.

- Read about the Options in the documentation [Wielemaker, 2008].

Reading terms

- Predicates for reading terms:

  read_term(Stream, Term, Options),
  read_term(Term, Options), using the current input stream,
  read(Stream, Term) like the above, without the options,
  read(Term) like the above, from current input.

- Read about the Options in the documentation [Wielemaker, 2008].

- The following example illustrates the use of variable_names, variables , singletons :

```
?- read_term(Term,[variable_names(Vars),
singletons(S),variables(List)]).
|     f(X, X, Y, Z).
Term = f(_G359, _G359, _G361, _G362),
Vars = ['X'=_G359, 'Y'=_G361, 'Z'=_G362],
S = ['Y'=_G361, 'Z'=_G362],
List = [_G359, _G361, _G362].
```

Writing terms

- Predicates for writing terms include:

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),
  write_term(Term, Options),

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),
  write_term(Term, Options),
  write(Stream, Term),

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),
  write_term(Term, Options),
  write(Stream, Term),
  write(Term),

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),
  write_term(Term, Options),
  write(Stream, Term),
  write(Term),

  the variants being analogous to the ones for reading terms.

Writing terms

- Predicates for writing terms include:

  write_term(Stream, Term, Options),
  write_term(Term, Options),
  write(Stream, Term),
  write(Term),

  the variants being analogous to the ones for reading terms.
- For options, other predicates for writing terms, consult the documentation.

Other I/O predicates

- Other I/O predicates include predicates for reading/writing characters/bytes:

Other I/O predicates

- Other I/O predicates include predicates for reading/writing characters/bytes:
  - get_char, peek_char, put_char, put_code, get_code, peek_code, get_byte, peek_byte, put_byte.

Other I/O predicates

- Other I/O predicates include predicates for reading/writing characters/bytes:
  - get_char, peek_char, put_char, put_code, get_code, peek_code, get_byte, peek_byte, put_byte.
- Other predicates:

Other I/O predicates

- Other I/O predicates include predicates for reading/writing characters/bytes:
  - get_char, peek_char, put_char, put_code, get_code, peek_code, get_byte, peek_byte, put_byte.
- Other predicates:
  - current_input, current_output, set_input, set_output, flush_output, at_the_end_of_stream, nl, etc.

Other I/O predicates

- Other I/O predicates include predicates for reading/writing characters/bytes:
  - get_char, peek_char, put_char, put_code, get_code, peek_code, get_byte, peek_byte, put_byte.
- Other predicates:
  - current_input, current_output, set_input, set_output, flush_output, at_the_end_of_stream, nl, etc.
- Consult the documentation for the details of the syntax.

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

- In Prolog, one can define new operators by specifying:

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

- In Prolog, one can define new operators by specifying:
  - the **position**: whether the operator is prefix (default), infix or postfix,

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

- In Prolog, one can define new operators by specifying:
  - the **position**: whether the operator is prefix (default), infix or postfix,
  - the **precedence**: to decide which operator applies first (e.g. is 2+3*4 (2+3)*4 or 2+(3*4)?), lower precedence binds the strongest, precedences are between 1 and 1200,

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

- In Prolog, one can define new operators by specifying:
  - the **position**: whether the operator is prefix (default), infix or postfix,
  - the **precedence**: to decide which operator applies first (e.g. is 2+3*4 (2+3)*4 or 2+(3*4)?), lower precedence binds the strongest, precedences are between 1 and 1200,
  - the **associativity**: e.g. is 8/2/2 (8/2)/2 or is it 8/(2/2)?

Operators in Prolog

- The usual way to write a Prolog predicate is to put it in front of its arguments:

```
functor(arg1, arg2, ...)
```

- However, there are situations where having a different position can make the programs easier to understand:

```
X is_father_of Y
```

- In Prolog, one can define new operators by specifying:
  - the **position**: whether the operator is prefix (default), infix or postfix,
  - the **precedence**: to decide which operator applies first (e.g. is 2+3*4 (2+3)*4 or 2+(3*4)?), lower precedence binds the strongest, precedences are between 1 and 1200,
  - the **associativity**: e.g. is 8/2/2 (8/2)/2 or is it 8/(2/2)?
  - Note that logic predicates (i.e. those expressions that evaluate to "true" or "false" are not associative in general). Consider: $3 = 4 = 3$, and suppose it were left associative, then $(3 = 4)$ $= 3$ evaluates to "false" $= 3$, which changes the type of the arguments.

Operator syntax specifiers

Specifier	**Meaning**
fx	Prefix, not associative.
fy	Prefix, right-associative.
xf	Postfix, not associative.
yf	Postfix, left-associative.
xfx	Infix, not associative (like $=$).
xfy	Infix, right associative (like comma in compound goals)
yfx	Infix, left associative (like $+$).

Commonly predefined Prolog operators

Priority	Specifier	Operators
1200	xfx	:-
1200	fx	:- ?-
1100	xfx	;
1050	xfy	->
1000	xfy	,
900	fy	not
700	xfx	= \= == \== @< is = =<
500	yfx	+ -
400	yfx	* / // mod
200	xfy	^
200	fy	-

Example

```
%note the syntax of declaring the new operator:

:- op(100, xfx, is_father_of).

 michael is_father_of kathy.
X is_father_of Y :- male(X), parent(X, Y).

?- X is_father_of kathy.

X = michael .
```

- Read: the paper [Covington, 1989, Covington et al., 1997].
- Read: Section 6.6, of [Covington et al., 1997].
- Read: Sections 2.2, 2.3, 2.6, 2.10, 2.12, A.7 of [Covington et al., 1997].
- Try out the examples in Prolog.

Covington, M. (1989).
Efficient Prolog: A Practical Guide.
Technical Report Research Report AI-1989-08, University of Georgia, Athens, Georgia.

Covington, M., Nute, D., and Vellino, A. (1997).
Prolog Programming in Depth.
Prentice Hall, New Jersey.

Wielemaker, J. (1990–2008).
SWI-Prolog 5.6.60 Reference Manual.
University of Amsterdam.