МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение

высшего образования

«**Вятский государственный университет**»

**(ФГБОУ ВО «ВятГУ»)**

Факультет автоматики и вычислительной техники

КАФЕДРА ЭЛЕКТРОННЫХ ВЫЧИСЛИТЕЛЬНЫХ МАШИН

Методические указания

по дисциплине: «Логическое программирование»

TURBO PROLOG И SWI-PROLOG

Киров, 2017

**Методические указания по выполнению лабораторных работ по дисциплине «Логическое программирование».**

Для выполнения лабораторной работы необзодимо скачать SWI-пролог 7.2.3 для Windovs и MacOS, а так же полную документацию можно по ссылке: <http://www.swi-prolog.org/download/stable>

Более подробно о стандартных предикатах и предикатах подключаемых библиотек SWI пролога можно узнать на официальном сайте: <http://www.swi-prolog.org/pldoc/man?section=summary>

Turbo Prolog - язык и система логического программирования, разработанные фирмой Borland в начале 80-х гг. Эта версия работает с дисковой операционнной системой (ДОС).

Turbo Prolog берет свои истоки из классического языка Prolog, автором которого являются Алэн Колмероэ и другие члены "группы искусственного интеллекта" Марсельского университета (Франция), реализованного впервые в 1973 г.

Turbo Prolog имеет существенные отличия от классического языка Prolog:

• Строгая типизация данных.

• Отсутствие возможности рассматривать правила как данные, т.е. добавлять и удалять их во время работы.

• Невозможность определять операции.

Turbo Prolog является компилированным языком, в отличии от многих других аналогов и отличается высокой скоростью трансляции и выполнения (что достигнуто как раз отходом от классического языка Prolog).

Также в данной системе добавлены многие интересные возможности, характерные только для этой реализации, такие как поддержка многооконного интерфейса и графики.

SWI-Prolog — это [свободная](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5) (открытая) реализация [языка программирования](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F)[Prolog](https://ru.wikipedia.org/wiki/Prolog), часто используемая для преподавания и приложений [Semantic Web](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D0%BC%D0%B0%D0%BD%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BF%D0%B0%D1%83%D1%82%D0%B8%D0%BD%D0%B0). Эта реализация предоставляет богатый набор возможностей, библиотеки для [constraint logic programming](https://ru.wikipedia.org/w/index.php?title=Constraint_logic_programming&action=edit&redlink=1), [многопоточности](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D0%BF%D0%BE%D1%82%D0%BE%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C), [юнит-тестирования](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D1%83%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D1%82%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), [GUI](https://ru.wikipedia.org/wiki/GUI), интерфейс к языку программирования [Java](https://ru.wikipedia.org/wiki/Java), [ODBC](https://ru.wikipedia.org/wiki/ODBC) и т. д., поддерживает [литературное программирование](https://ru.wikipedia.org/wiki/%D0%93%D1%80%D0%B0%D0%BC%D0%BE%D1%82%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), содержит реализацию [веб-сервера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), библиотеки для [SGML](https://ru.wikipedia.org/wiki/SGML), [RDF](https://ru.wikipedia.org/wiki/Resource_Description_Framework), RDFS, средства разработчика (включая [IDE](https://ru.wikipedia.org/wiki/Integrated_Development_Environment) с графическими [отладчиком](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BB%D0%B0%D0%B4%D1%87%D0%B8%D0%BA) и[профилировщиком](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%84%D0%B8%D0%BB%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0))), и обширную документацию.

SWI Prolog — довольно популярная система, в основном благодаря удобной среде и переносимой библиотеке для создания графического интерфейса. SWI-Prolog, как почти все реализации языка, по большей части реализует [Edinburgh Prolog](http://progopedia.ru/dialect/edinburgh-prolog/), но также содержит отдельные элементы [ISO Prolog](http://progopedia.ru/dialect/iso-prolog/).

SWI-Prolog включает в себя быстрый компилятор, профилировщик, набор библиотек и удобный интерфейс для подключения [C](http://progopedia.ru/language/c/)-модулей. SWI-Prolog работает на платформах Unix, Windows, и Macintosh. Аналоги предикатов SWI-пролога для турбо пролога представлены в таблице1.

Таблица1- Сравнение предикатов Турбо Пролог и SWI-Prolog

|  |  |
| --- | --- |
| Турбо Пролог | SWI-Prolog |
| Типы данных | |
| *symbol* Последовательность букв, цифр н знаков подчеркивания, которая начинается со строчной буквы или заключена в кавычки  *string* Любая последовательность символов, которая заключена в кавычки  *char* Отдельный символ, заключенный в апострофы  *integer* Целое число в диапазоне от -32768 до 32767  *real*Любое число, может быть представлено в экспоненциальной форме  *file* Имя файла  Проиллюстрируем вышесказаниое примерами:  *symbol* person, al,b2,autonummer, "Prolog"  *string* "превосходный сыр","123/456","кочан"  *char* 'A','3','?','\*'  *integer* 5,17,-1234  *real* 123,0.876,-50000,1.2e+6  *file* lager, einkauf,BASISl | Поддерживает динамическую типизацию данных.  Если имя объекта начинается со строчной буквы, то этот объект имеет известное значение, то есть это константа. Объекты с известными или постоянными значениями должны начинаться со строчных букв, если они не являются числами.  Слова, начинающиеся с прописных букв, являются переменными языка SWI-Prolog.  Внутренние унификационные подпрограммы означивают переменные.  Означенные переменные и константы имеют значения, «известные» SWI-Prolog.  Свободные или неозначенные переменные значений не имеют.  Переменные могут быть объектами предиката как в утверждениях, так и в подцелях.  Числа  ?- A = 12, B = -0.5e4.  A = 12,  B = -5000.0.    ?- number($A), number($B).  true. % показываем, что тип переменных - числа  Атомы  ?- A = abc, B = 'Hello World'.  A = abc,  B = 'Hello World'.    ?- atom($A), atom($B).  true. % показываем, что тип переменных - атомы  Строки  ?- S = "Привет мир".  S = [1055, 1088, 1080, 1074, 1077, 1090, 32, 1084, 1080|...].  Видно, что строки являются списками кодов символов, т.е. к ним применимы все те же операции что и к спискам.  Списки  ?- A=[], B=[a, foo, 123, [[[[[1,2,42]],bar]]], "Привет", A], C=[A,B].  A = [], % пустой список    B = [a, foo, 123, [[[[[1|...]], bar]]], [1055, 1088, 1080, 1074|...], []],  C = [[], [a, foo, 123, [[[[...]|...]]], [1055, 1088|...], []]]  Списки могут быть:  Разнородными (содержать любые комбинации выше (и ниже) перечисленных типов)  Вложенными  Структуры  ?- A = aaa(bb), B = aaa(bbbbbb, 123, [456, c]), C = ccc(ddd(eee), fff, g(h(i(j(kkkkk))))).  A = aaa(bb),  B = aaa(bbbbbb, 123, [456, c]),  C = ccc(ddd(eee), fff, g(h(i(j(kkkkk)))))  Структура в прологе представляется функтором (имя структуры, то что до скобок) и параметрами (то что в скобках). Число параметров называется арностью функтора. Как видим, структуры тоже могут быть вложенными. |
| Арифметические и логические предикаты | |
| sin(X) Синус  cos(X) Косинус  tan(X) Тангенс  arctan(X) Арктангенс  abs(X) Абсолютная величина  exp(X) Экспонента  ln(X) Натуральный логарифм  log(X) Десятичный логарифм  sqrt(X) Квадратный корень  bitand(A,B,E) Логическое И (битовое)  bitor(A,B,E) Логическое ИЛИ (битовое)  bitxor(A,B,E) Исключающее ИЛИ (битовое)  bitnot(A,B) Логическое НЕ (битовое)  bitJeft(A,N,E) Сдвиг на N битов влево  bitright(A,N,E) Сдвиг на N битов вправо | abs(X) - абсолютная величина X;  exp(X) - экспонента;  acos(X) - арккосинус X;  ln(X) - логарифм натуральный;  asin(X) - арксинус X;  log(X) - логарифм десятичный;  atan(X) - арктангенс X;  sin(X) - синус X;  cos(X) - косинус X;  tan(X) - тангенс X;  sqrt(X) - квадратный корень из X.  round(X,N) - X округляется до N десятичных цифр, где N - целое число между 0 и 15. |
| Список операторов | |
| + Сложение  - Вычитание  \* Умножение  / Деление  mod Абсолютная величина  div Целочисленное деление  not HE (высший приоритет)  and И (средний приоритет)  or ИЛИ (низший приоритет) | *Операторы арифметических выражений:*  X+Y - сложение.  X\*Y - умножение;  X-Y - вычитание.  X^Y - возведение в степень;  X/Y - деление с вещественным результатом.  X//Y - деление целых чисел с целым результатом.  -X - унарный минус (меняет знак X).  *Поразрядные операторы для целых чисел:*  X/\Y - конъюнкция (And); X\/Y - дизъюнкция(OR);  \(X) - отрицание (NOT);  X<<Y - сдвиг X влево на Y позиций;  X>>Y - сдвиг X вправо на Y позиций;  [X] - вычисляется как X.  X mod Y - возвращает остаток от деления X на Y |
| Операции отношения | |
| > Больше  < Меньше  = Равно  >= Больше или равно  <= Меньше или равно  <> Не равно | = Унификация (присваивание значения несвязанной переменной)  <, =<, >=, > Арифметические (только для чисел) операции сравнения  =:= Арифметическое равенство  =\= Арифметическое неравенство  is Вычисление арифметического выражения (например, A is 5 + 2)  @<, @=<, @>=, @> Операции сравнения для констант и переменных любого типа (чисел, строк, списков и т.д.)  == Равенство констант и переменных любого типа  \== Неравенство констант и переменных любого типа |
| Основные встроенные предикаты | |
| asserta  Функция: Добавляет факт (в начало).  Формат: asserta(Факт)  Комментарий: Заносит факт в базу данных перед другими фактами. В результате данный факт будет добавлен в начало базы данных. | |
| assertz  Функция: Добавляет факт (в конец).  Формат: assertz(Факт)  Комментарий: Заносит факт в базу данных за другими фактами. В результате данный факт будет добавлен  в конец базы данных. | |
| attribute  Функция: Устаиовка/опрос атрибута.  Формат: attribute(Атрибут)  Комментарий: Устанавливает значение атрибута, определяющее цвет фона текущего окна. Значение атрибута определяется параметром. |  |
| back  Функция: Продвигает перо в обратном направлении.  Формат: bаск(Ша г)  Шаг: *integer(oт 0 до 31999)*  Комментарий: Продвигает перо назад на заданное количество шагов. Направление может быть предварительно определено предикатами *left* или *right.* Экран должен на­ходиться в графическом режиме. |  |
| Beep  Функция: Производит звуковой сигнал |  |
| bios  Функция: Объявляет прерывание для вызова процедур BIOS.  Формат: bios(Прерывание, ВхРегистр, ВыхРегнстр)  Комментарий: Обеспечивает формирование прерывания с заданным номером. Регистры получают значения, уста­новленные параметром "ВхРегистр". После обработки пре­рывания содержимое регистров связывается с параметром "ВыхРегистр". Параметры "ВхРегистр" и "ВыхРегнстр" должны принадлежать домену *regdom,* который определяется сле­дующим образом:  *regdom=reg(AX, ВХ, CX, DX, SI, DI, DS, ES)*  где *АХ, ВХ ... ES* должны иметь тип *integer.* |  |
| char\_int  Функция: Преобразует представления символов и чисел.  Формат: char\_int(Символ, Число)  Комментарий:  Связывает параметр "Число" с кодом ASCII параметра "Символ".  Связывает параметр "Символ" с симво­лом, код которого определяется параметром "Число".  Выполняется успешно, если код, опре­деляемый параметром "Число", является ASCII - кодом символа, определяемого параметром "Символ". | number\_chars/2 Convert between number and one-char atoms - Преобразование между числом и односимвольным атомом  number\_codes/2 Convert between number and character codes - Преобразование между числом и символьным кодом |
| clearwindow  Функция: Очистка текущего окна. |  |
| Closefile  Функция: Закрывает файл.  Формат: Closefile(ИмяФайла)  Комментарий: Закрывает данный файл. Имя файла не должно быть заключено в кавычки. Выполняется успеш­но, даже если данный файл перед этим не был открыт. | close/1 - Закрыть поток(файл)  close/2 - Закрыть поток/файл (принудительный) |
| concat  Функция: Склеивает две строки.  Формат: concat(Строка1,Строка2,СтрокаЗ)  Комментарий: Склеивает строки, заданные первым и вторым параметрами, и связывает результат с третьим па­раметром. | string\_concat Конкатенация двух строк  atom\_concat Конкатенация двух атомов  append(List1,List2, List3) Успешен, когда список List3 унифицируем с объединением списков List1 и List2. Все аргументы могут быть свободными переменными. Результат унификация соответствующих списков.  append(ListOfLists, List) Успешен, когда объединение списка списков ListOfLists унифицируем со списком List. Результат: унификация соответствующих списков. |
| consult  Функция: Добавляет текстовый файл в базу  данных.  Формат: consult(имяФайлаDOS)  Комментарий: Записывает в базу данных текстовый файл. Текстовый файл может быть, например, создан в ре­зультате выполнения предиката *save.* Этот файл содержит факты, которые должны быть описаны в разделе *database.* Выполнение предиката *consult* не будет успешным, если в файле имеются синтаксические ошибки.  Факты из текстового файла дополняют набор уже существующих фактов, а не заменяют их. | **consult**(:File) - Чтение файла |
| cursor  Функция: Позиционирование курсора.  Формат: cursor(Cтрока, Столбец)  Комментарий:  Передвигает курсор в текущем окне в позицию, заданную номерами строки и столбца. Номер строки может находиться в диапазоне от 0 до 24, номер столбца - от 0 до 79. Координаты левого верхнего угла экрана - *(0,0).*  Связывает номера строки и столбца, определяющие позицию курсора, с соответствующими пара­метрами. |  |
| cursorform  Функция: Определение курсора.  Формат: cursorform(НачСтрока, КонСтрока)  Комментарий: Определяет высоту и вертикальную координату курсора внутри области, занимаемой одним символом. Символ перекрывает 14 строк развертки. Таким образом, параметры могут принимать значения в диапазоне от 1 до 14. |  |
| cut(!)  Функция: Предотвращение бектрекинга.  Формат: !  Комментарий: Предотвращает использование механиз­ма бектрекинга. Записывается как восклицательный знак. | |
| date  Функция: Установка и считывание даты.  Формат: date(Год, Месяц, День) | day\_of\_the\_week Определяет день недели из даты  date\_time\_stamp Преобразовать дату во время  date\_time\_value Извлечение даты из структуры даты по ключу |
| deletefile  Функция: Удаление файла.  Формат: deletefile(ИмяФайлаDOS)  Комментарий: Удаляет заданный файл DOS. Параметр может содержать идентификатор накопителя, но не должен указывать путь. | delete\_file  Функция: Удаление файла.  Формат: delete\_file(ИмяФайла)  Комментарий: Удаляет заданный файл. |
| display  Функция: Показывает строку.  Формат: display(GrpoKa)  Комментарий: Показывает заданную строку в текущем окне. Строка при этом не может быть изменена. | write/1 - Записать утверждение  write/2 - Записать утверждение для потоковой передачи  writeln/1 - Записать утверждение с последующим переводом строки  writeln/2 - Записать утверждение за которым следует символ новой строки в поток  write\_canonical/1 - Записать утверждение в кавычки, игнорируя операторы  write\_canonical/2 - Записать утверждение в кавычки, игнорируя операторы в поток  nl/0 - Генерация новой строки  nl/1 - Генерация новой строки на поток |
| edit  Функция: Вызов редактора Турбо-Пролога.  Формат: еdit(ВхСтрока, ВыхСтрока)  Комментарий: Строка, связанная с первым парамет­ром, показывается в текущем окне. Пользователь может изменять эту строку с помощью команд редактора Тур­бо-Пролога. Измененная строка будет связана со вторым параметром. | edit - Редактировать текущий файл |
| eof(ИмяФайлаDOS)  Выполняется успешно, если указатель текущей позиции файла указывает на конец файла, и завершается неудачно в противном случае. | at\_end\_of\_stream/0 - Тест на конец файла на входе  at\_end\_of\_stream/1 - Тест на конец файла в потоке |
| existfile(ИмяФайлаDOS)  Выполняется успешно, если заданный файл присутствует в текущем каталоге, и завершается неудачно в противном случае. | exists\_file/1 - Проверить существование файла |
| fail  Осуществляет вынужденное неудачное завершение выполнения предиката | |
| field\_str  Функция: Запись или чтение строки.  Формат: field\_str(Строка, Столбец, Длина,СтрСимволов)  Комментарий: Записывает строку, связанную с параметром "СтрСимволов", в поле, определяемое длиной и но­мерами строки и столбца. Если строка длиннее, чем задан­ное поле, то записывается только начало строки. Если строка короче, то оставшиеся позиции поля заполняются пробелами. | peek\_string/3 - Прочитать строку без удаления  read\_string/3 - Прочитать числа символов в строку  read\_string/5 - ПроПрочитатьстроки по разделителю  read\_term/2 - Прочитать утверждение с параметрами  read\_term/3 - Прочитать утверждение с параметрами из потока  read\_term\_from\_atom/3 - Прочитать утверждение с параметрами из атома |
| filepos  Функция: Установка или чтение позиции указателя.  Формат: Filepos(ИмяФайла, Позиция, Режим)  Комментарий: Устанавливает указатель данного фай­ла на заданную позицию. |  |
| file\_str  Функция: Читает строку из файла.  Формат: file\_str(ИмяФайлаDOS, Строка)  Комментарий: Читает строку из заданного файла и связывает ее с параметром "Строка". Максимально допусти­мый размер строки - 64 К. |  |
| findall  Функция: Собирает значения, возникающие в про-  цессе бектрекинга, в список.  Комментарий: С параметром "Список" связываются все значения, определяемые первым параметром, которые возникают, когда заданный вторым параметром предикат выполняется в процессе бектрекинга. | |
| flush  Функция: Очищает содержимое буфера.  Формат: flush(ИмяФайла)  Комментарий: Содержимое внутреннего файлового бу­фера пересылается в заданный файл. С помощью предиката *flush* можно, например, организовать передачу данных на стандартный интерфейс до того, как буфер будет за­полнен. |  |
| free  Функция: Проверяет, свободна ли переменная.  Формат: fгее(Переменная)  Комментарий: Выполняется успешно, если заданная переменная не связана. | numbervars/3 - Число свободных переменных утверждения  numbervars/4 - Число свободных переменных утверждения  ground/1 - Убедитесь, что термин содержит не свободные переменные |
| frontchar  Функция: Разделяет заданную строку на две части: первый символ и оставшаяся часть строки.  Формат: frontchar(Строка, ПервСимвол, Остаток)  Комментарий: Связывает со вторым параметром пер­вый символ строки, заданной параметром "Строка", а с параметром "Остаток" - оставшуюся часть строки.  Связывает с параметром "Остаток" ос­тавшуюся часть заданной первым параметром строки, полу­ченную после удаления из нее символа, связанного с па­раметром "ПервСимвол"  Связывает с параметром "ПервСимвол" первый символ заданной строки, если с третьим парамет­ром связан остаток строки.  Выполняется успешно, если заданная первым параметром строка может быть получена склеивани­ем символа, заданного вторым параметром, со строкой, связанной с параметром "Остаток". | split\_string/4 - Разбить строку на подстроки  sub\_atom/5 - Извлечь подстроку из атома  sub\_string/5 - Извлечь подстроку из строки |
| fronttoken  Функция: Разделяет заданную строку на лексему и остаток.  Формат: fronttoken( Строка, Лексема, Остаток)  Комментарий: Разделяет строку, заданную парамет­ром "Строка", на лексему и остаток. Лексема связывается со вторым параметром, остаток - с третьим.  Лексема - это последовательность символов, имею­щих смысл.  Связывает с третьим параметром оста­ток заданной строки, полученный при удалении из нее лексемы, заданной вторым параметром.  Связывает с параметром "Лексема" лексему из заданной строки.  Выполняется успешно, если строку, связанную с параметром "Строка", можно получить в ре­зультате склеивания лексемы и остатка  Связывает с первым параметром стро­ку, которая получена в результате склеивания лексемы и остатка. |
| Graphics – устанавливает графический режим |  |
| inkey  Функция: Читает символ.  Формат: inkey(Символ)  Комментарий: Читает символ со стандартного уст­ройства ввода. В отличие от предиката readchar выполне­ние программы не прерывается. Поэтому inkey применяют главным образом для организации циклов ожидания. Преди­кат inkey не выполняется, если не введен ни один символ. | get/1 - Прочитать первый непустой символ  get/2 - Прочитать первый непустой символ из потока  get\_char/1 - Прочитать следующий символ как atom (ISO)  get\_char/2 - Прочитать следующий символ из потока (ISO)  get\_code/1 - Прочитать следующий символ (ISO)  get\_code/2 - Прочитать следующий символ из потока (ISO)  get0/1 - Прочитать следующий символ  get0/2 - Прочитать следующий символ из потока  peek\_char/1 - Чтение символов без удаления  peek\_char/2 - Чтение символов без удаления |
| keypressed  Функция: Проверяет, нажата ли клавиша.  Формат: keypressed  Выполняется успешно, если нажата некоторая клави­ша. В отличие от предиката inkey с помощью keypressed можно установить, нажата ли клавиша, не читая при этом введенный с клавиатуры символ. |  |
| left  Функция: Поворачивает перо влево.  Формат: left (Угол)  Комментарий: Поворачивает перо на указанный угол против часовой стрелки. Экран должен находиться в гра­фическом режиме. |  |
| line  Функция Рисует линию.  Формат: line(Строка1, Столбец1,Строка2, Стол6ец2,  Цвет)  Комментарий: Рисует линию при работе в графичес­ком режиме. Координаты первой точки задаются первыми двумя параметрами, координаты второй точки - третьим и четвертым параметрами. Значение "Цвет" задается в со­ответствии с приведенной ниже таблицей. Номера строк и столбцов должны находиться в диапазоне от 0 до 31999. Предикат line не выполняется, если заданы недопустимые позиции. |  |
| Membyte  Функция: Запоминает или считывает байт.  Формат: membyte(Сагмент, Смещение, Байт)  Комментарий: Записывает по заданному адресу зна­чение параметра "Байт". Адрес вычисляется по формуле адрес=сегмент\*16+смещение. | get\_byte/1 - Прочитать следующий байт (ISO)  get\_byte/2 - Прочитать следующий байт из потока (ISO)   |  | | --- | | peek\_byte/1 - Прочитать байт без удаления | | peek\_byte/2 - Прочитать байт без удаления | |
| nl  Функция: Перевод строки.  Формат: nl  Происходит перевод строки, такой, как при посылке на текущее устройство вывода символа возврата кареткн | |
| not  Функция: Отрицание.  Формат: not (Атом)  Комментарий: Выполняется успешно, если заданный атом представляет собой цель, которая не достигается. | |
| openappend  Функция: Открывает файл для дополнения.  Формат: oprnappend(ИмяФанла, ИмяФайла DOS)  Комментарий: Открывает файл для дополнения. Связы­вает заданное первым параметром имя с именем файла DOS. | |  |  | | --- | --- | | load\_files/1 - Загружать исходные файлы | | | load\_files/2 - Загружать исходные файлы с параметрами | | | open/3 - Открыть файл (создание потока) | | open/4 - Открыть файл (создание потока) |   prolog\_load\_file/2 - (хук) Программа загрузки файлов / 2  append/1 - Добавлять в файл |
| openmodify  Функция: Открывает файл для чтения/записи.  Формат: openmodifу(ИмяФайла, ИмяФайлаDOS)  Комментарий: Открывает файл для чтения или записи. Связывает заданное первым параметром имя с именем файла DOS. |
| openread  Функция: Открывает файл для чтения.  Формат: ореnread(ИмяФaнл а, ИмяФайлаDOS)  Имя Файлa: file ИмяФайлаDOS: string  1. Прототип: (i,i)  Комментарий: Открывает файл для чтения. Связывает заданное первым параметром имя с именем файла DOS. |
| openwrite  Функция: Открывает файл для записи.  Формат: openwritе(ИмяФайла, ИмяФайлаDOS)  ИмяФайла: file ИмяФайлаDOS: string  1. Прототип: (i,i)  Комментарий: Открывает файл для чтения. Связывает заданное первым параметром имя с именем файла DOS. |
| pencolor  Функция: Устанавливает цвет линии, проводимой пером.  Формат: репСо1оr(Цвет)  Комментарий: Определяет цвет линии, проводимой пером. Должен быть установлен графический режим экра­на. |  |
| pendown  Функция: Активирует перо. |  |
| penpos  Функция: Устанавливает или читает позицию пера.  Формат: penpos(Строка, Столбец, Направление)  Комментарий: Устанавливает перо в заданную пози­цию. Номера строки и столбца задаются в пикселах. На­правление задается в градусах (0 градусов - вниз). До­пустимые значения номеров строки и столбца - от 0 до 31999.  Читает позицию пера. Параметры свя­зываются с номерами строки, столбца и направлением. |  |
| penup  Функция: Деактивирует перо. |  |
| readchar  Функция: Читает символ с текущего устройства ввода.  Формат: readchar(СимвПеременная)  Комментарий: Читает символ с текущего устройства ввода и связывает его с заданной переменной. В отличие от inkey устанавливает режим ожидания ввода. | get/2 - Прочитать первый непустой символ из потока  read/1 - Прочитать утверждение Пролог  read/2 - Прочитать утверждение Пролог из потока  read\_clause/3 - Прочитать статью из потока  read\_history/6 - Прочитать используя историю замещения  read\_link/3 - Прочитать символическую ссылку  read\_pending\_input/3 - Извлечь буферизованный ввод из потока  read\_string/3 - Прочитать числа символов в строку  read\_string/5 - ПроПрочитатьстроки по разделителю  read\_term/2 - Прочитать утверждение с параметрами  read\_term/3 - Прочитать утверждение с параметрами из потока  read\_term\_from\_atom/3 - Прочитать утверждение с параметрами из атома |
| readint  Функция: Читает целое число.  Формат: readint (ЦелаяПерем)  Комментарий: Читает целое число с текущего уст­ройства ввода и связывает его с заданной переменной. |
| readln  Функция: Читает строку.  Формат: readln(Строка)  Комментарий: Читает строку с текущего устройства ввода и связывает ее с заданной переменной. В качестве конца строки используется Carriage Return (CR - возврат каретки). Обычно чтение производится с клавиатуры. |
| readreal  Функция: Читает действительное число.  Формат: readreal(ВещПерем)  ВещПерем: real  1. Прототип: (о)  Коммеитарий: Читает действительное число с теку­щего устройства ввода и связывает его с заданной пере­менной. Обычно чтение производится с клавиатуры. |
| Readterm  Читает любые объекты.  readterm(Область,Терм)  Читает объект, который был записан предикатом write. C помощью reddterm осуществляется доступ к фактам в файле. | Read\_term(term, option) – читает терм из потока, под контролем опций |
| retract  Функция: Удаляет факт из базы данных.  Формат: retract(Факт)  Комментарий: Удаляет из базы данных первый факт, который отождествляется с заданным фактом. | |
| right  Поворачивает перо вправо на установленный угол. |  |
| Save(ИмяФайла)  Комментарий: Записывает динамическую базу данных на дискету или жесткий диск. |  |
| Scroll  Функция: Сдвиг содержимого текущего окна,  Формат: scroll(ЧислоСтрок,ЧислоСтолбцов)  Комментарий: Сдвигает содержимое текущего на заданное число строк и столбцов. |  |
| Storage  Функция: Определяет размер имеющейся памяти.  Формат: storage(Стек, ДинПам, Остаток)  Комментарий: Связывает с соответствующими пара­метрами размеры имеющихся в распоряжении областей памя­ти. |  |
| str\_char  Функция: Преобразует строку в символ или на-  оборот.  Формат: str\_char( Строка, Символ) | string(*Term*) – меняет тип терма на строковый |
| str\_int  Функция: Преобразует строку в целое число  или наоборот.  Формат: str\_int(Строкa, ЦелоеЧисло) | atom Проверка типа атома  atom\_chars Преобразование между атомом и списком символов  atom\_codes Преобразование между атомом и списком кодов символов  atom\_length Определить длину атома  atom\_number Преобразование между атомом и числом  atom\_string Преобразование между атомом и строки |
| Str\_len  Функция: Определяет длину строки.  Формат: str\_\_len( Строка, Длина) | string\_length(строка, длина) - Определяем длину строки |
| str\_real  Функция: Преобразует строку в действительное  число или наоборот.  Формат: str\_геа1(Строка,ДействительноеЧисло) |  |
| Text - Устанавливает текстовый режим. |  |
| time  Функция: Устанавливает или считывает системное время.  Формат: time(чac, Мин, Сек, СотаяСек) | get\_time/1 - Получить текущее время  format\_time/3 - C STRFTIME (), похожий формат дата / время |
| Trace - Активирует трассировку | |
| upper\_lower  Функция: Преобразует прописные буквы в строчные или наоборот.  Формат: upper\_lower(Строка1, Строка2) |  |
| window\_attr  Функция: Определяет атрибуты текущего окна.  Формат: window\_attr( Атрибут)  Комментарий: Определяет атрибуты текущего окна. |  |
| window\_str  Функция: Записывает строку в текущее окно или считывает строку из текущего окна. |  |
| Write  Функция: Производит запись на текущее устройство вывода.  Формат: write(A1,A2,A3,...)  Комментарий: Записывает заданные значения на те­кущее устройство вывода. | |
| writef  Функция: Осуществляет форматный вывод.  Формат: изводит форматный вывод на текущее устройство вывода. | |

Полный список встроенных придикатов SWI-пролога представлен в Приложении Б.

Более подробно о стандартных предикатах и предикатах подключаемых библиотек SWI пролога можно узнать на официальном сайте: <http://www.swi-prolog.org/pldoc/man?section=summary>

Скачать SWI-пролог 7.2.3 для Windovs и MacOS, а так же полную документацию можно по ссылке: <http://www.swi-prolog.org/download/stable>

Приложение Б

(Обязательное)

Встроенные придикаты SWI-пролога

|  |  |  |
| --- | --- | --- |
| Предикат | Английское описание | Русское описание |
| @/2 | Call using calling context | Вызов с помощью контекста вызова |
| !/0 | Cut (discard choicepoints) | Вырезать (отменить точки выбора) |
| ,/2 | Conjunction of goals | Конъюнкции целей |
| ->/2 | If-then-else | Если-то-иначе |
| \*->/2 | Soft-cut | Упрощенное отсечение |
| ./2 | Consult. Also functional notation | Консультации. Также функциональная нотация |
| :</2 | Select keys from a dict | Выбор ключей из справочника |
| ;/2 | Disjunction of two goals | Дизъюнкция (сложение) двух целей |
| </2 | Arithmetic smaller | Арифметическое уменьшение |
| =/2 | True when arguments are unified | Значение правда, когда аргументы объединяются |
| =../2 | ``Univ.'' Term to list conversion | ' Univ.'' Утверждение для списка коэффициентов |
| =:=/2 | Arithmetic equality | Арифметическое равенство |
| =</2 | Arithmetic smaller or equal | Арифметическое меньше или равно |
| ==/2 | Test for strict equality | Тест на строгое равенство |
| =@=/2 | Test for structural equality (variant) | Тест на структурное равенство (вариант) |
| =\=/2 | Arithmetic not equal | Арифметическое не равно |
| >/2 | Arithmetic larger | Арифметическое больше |
| >=/2 | Arithmetic larger or equal | Арифметическое больше или равно |
| >:</2 | Partial dict unification | Частичное объединение справочников |
| ?=/2 | Test of terms can be compared now | Тест утверждений сравнимых прямо сейчас |
| @</2 | Standard order smaller | Стандартная сортировка по возрастанию |
| @=</2 | Standard order smaller or equal | Стандартная сортировка меньше или равно |
| @>/2 | Standard order larger | Стандартная сортировка по убыванию |
| @>=/2 | Standard order larger or equal | Стандартная сортировка больше или равен |
| \+/1 | Negation by failure. Same as not/1 | Отрицание отказом. Так же, как не/1 |
| \=/2 | True if arguments cannot be unified | Правда, если аргументы не могут быть объединены |
| \==/2 | True if arguments are not strictly equal | Правда, если аргументы не являются строго равны |
| \=@=/2 | Not structural identical | Не структурно идентичен |
| ^/2 | Existential quantification (bagof/3, setof/3) | Квантификация существования (bagof/3, setof/3) |
| |/2 | Disjunction in DCGs. Same as | Сложение в DC-грамматиках. Аналогично; |
| /1 | DCG escape; constraints | DC-грамматика - побег ; ограничения |
| abolish/1 | Remove predicate definition from the database | Удалить определение предиката из базы данных |
| abolish/2 | Remove predicate definition from the database | Удалить определение предиката из базы данных |
| abort/0 | Abort execution, return to top level | Прервать выполнение, вернуться к верхнему уровню |
| absolute\_file\_name/2 | Get absolute path name | Получить полный путь |
| absolute\_file\_name/3 | Get absolute path name with options | Получить полный путь с опциями |
| access\_file/2 | Check access permissions of a file | Проверка права доступа к файлу |
| acyclic\_term/1 | Test term for cycles | утверждение испытания для циклов |
| add\_import\_module/3 | Add module to the auto-import list | Добавить модуль в список авто импорта |
| add\_nb\_set/2 | Add term to a не-backtrackable set | Добавить утверждение в не backtrackable набор |
| add\_nb\_set/3 | Add term to a не-backtrackable set | Добавить утверждение в не backtrackable набор |
| append/1 | Append to a file | Добавлять в файл |
| apply/2 | Call goal with additional arguments | Вызов цели с дополнительными аргументами |
| apropos/1 | library(online\_help) Search manual | Библиотека онлайн хелпа - поиск справки |
| arg/3 | Access argument of a term | Аргумент доступа утверждения |
| assoc\_to\_list/2 | Convert association tree to list | Преобразование дерева ассоциации в список |
| assert/1 | Add a clause to the database | Добавить пункт в базу данных |
| assert/2 | Add a clause to the database, give reference | Добавить статью в базу данных, дайте ссылку |
| asserta/1 | Add a clause to the database (first) | Добавить статью в базу данных (первый) |
| asserta/2 | Add a clause to the database (first) | Добавить статью в базу данных (первый) |
| assertion/1 | Make assertions about your program | Сделать утверждения о вашей программе |
| assertz/1 | Add a clause to the database (last) | Добавить статью в базу данных (в прошлом) |
| assertz/2 | Add a clause to the database (last) | Добавить статью в базу данных (последняя) |
| attach\_console/0 | Attach I/O console to thread | Приложить консоль ввода / вывода в поток |
| attribute\_goals/3 | Project attributes to goals | Проект атрибутов целей |
| attr\_unify\_hook/2 | Attributed variable unification hook | Приписывается переменная объединения хук |
| attr\_portray\_hook/2 | Attributed variable print hook | Приписывается переменная хук для печати |
| attvar/1 | Type test for attributed variable | Тип теста атрибутом переменной |
| at\_end\_of\_stream/0 | Test for end of file on input | Тест на конец файла на входе |
| at\_end\_of\_stream/1 | Test for end of file on stream | Тест на конец файла в потоке |
| at\_halt/1 | Register goal to run at halt/1 | Цель регистра для запуска на остановке/1 |
| atom/1 | Type check for an atom | Проверка типа для атома |
| atom\_chars/2 | Convert between atom and list of characters | Преобразование между атомом и списком символов |
| atom\_codes/2 | Convert between atom and list of characters codes | Преобразование между атомом и списком кодов символов |
| atom\_concat/3 | Contatenate two atoms | Contatenate два атома |
| atom\_length/2 | Determine length of an atom | Определить длину атома |
| atom\_number/2 | Convert between atom and number | Преобразование между атомом и числом |
| atom\_prefix/2 | Test for start of atom | Тест для начала атома |
| atom\_string/2 | Conversion between atom and string | Преобразование между атомом и строки |
| atom\_to\_term/3 | Convert between atom and term | Преобразование между атомом и утверждение |
| atomic/1 | Type check for primitive | Проверка типа для примитивных |
| atomic\_concat/3 | Concatenate two atomic values to an atom | Соединить две атомные значения атома |
| atomic\_list\_concat/2 | Append a list of atomics | Добавляем список Атомикс |
| atomic\_list\_concat/3 | Append a list of atomics with separator | Добавляем список Атомикс с сепаратором |
| atomics\_to\_string/2 | Concatenate list of inputs to a string | Соединить список входов в строку |
| atomics\_to\_string/3 | Concatenate list of inputs to a string | Соединить список входов в строку |
| autoload/0 | Autoload all predicates now | Автозагрузка всех предикатов сейчас |
| autoload\_path/1 | Add directories for autoloading | Добавление каталогов для автоматической загрузки |
| b\_getval/2 | Fetch backtrackable global variable | Извлечь backtrackable глобальную переменную |
| b\_set\_dict/3 | Destructive assignment on a dict | Пометить словарь на уничтожение |
| b\_setval/2 | Assign backtrackable global variable | Назначить backtrackable глобальную переменную |
| bagof/3 | Find all solutions to a goal | Найти все решения цель |
| between/3 | Integer range checking/generating | Integer проверки диапазона / генерирования |
| blob/2 | Type check for a blob | Проверка типа для blob-объекта |
| break/0 | Start interactive top level | Запустить интерактивный верхний уровень |
| break\_hook/6 | (hook) Debugger hook | (хук) Отладчик хук |
| byte\_count/2 | Byte-position in a stream | Байт-позиции в потоке |
| call/1 | Call a goal | Вызов цели |
| call/[2..] | Call with additional arguments | Вызов с дополнительными аргументами |
| call\_cleanup/3 | Guard a goal with a cleaup-handler | Защитить цель очисткой обработчика |
| call\_cleanup/2 | Guard a goal with a cleaup-handler | Защитить цель очисткой обработчика |
| call\_dcg/3 | As phrase/3 without type checking | Как выражение/3 без проверки типа |
| call\_residue\_vars/2 | Find residual attributed variables | Найти остаточные переменные, приписываемые |
| call\_shared\_object\_function/2 | UNIX: Call C-function in shared (.so) file | UNIX: Call C-функцию в общей (.so) файла |
| call\_with\_depth\_limit/3 | Prove goal with bounded depth | Доказать цель с ограниченной глубиной |
| call\_with\_inference\_limit/3 | Prove goal in limited inferences | Доказать цель в ограниченные выводы |
| callable/1 | Test for atom or compound term | Испытание на атом или составное утверждение |
| cancel\_halt/1 | Cancel halt/0 from an at\_halt/1 hook | Отменить привал / 0 от at\_halt / 1 Хук |
| catch/3 | Call goal, watching for exceptions | Вызов цели, наблюдая за исключением |
| char\_code/2 | Convert between character and character code | Преобразование между характером и кодом символа |
| char\_conversion/2 | Provide mapping of input characters | Обеспечить отображение входных символов |
| char\_type/2 | Classify characters | Классифицировать символы |
| character\_count/2 | Get character index on a stream | Получить индекс символа на поток |
| chdir/1 | Compatibility: change working directory | Совместимость: изменить рабочий каталог |
| chr\_constraint/1 | CHR Constraint declaration | CHR Constraint декларация |
| chr\_show\_store/1 | List suspended CHR constraints | Список приостановлено ограничения CHR |
| chr\_trace/0 | Start CHR tracer | Запустить CHR трейсер |
| chr\_type/1 | CHR Type declaration | Объявление типа CHR |
| chr\_notrace/0 | Stop CHR tracer | Остановить трассировщик CHR |
| chr\_leash/1 | Define CHR leashed ports | Определить CHR поводке порты |
| chr\_option/2 | Specify CHR compilation options | Указать параметры компиляции CHR |
| clause/2 | Get clauses of a predicate | Получить предложения предиката |
| clause/3 | Get clauses of a predicate | Получить предложения предиката |
| clause\_property/2 | Get properties of a clause | Получить свойства пункта |
| close/1 | Close stream | Закрыть поток |
| close/2 | Close stream (forced) | Закрыть поток (принудительный) |
| close\_dde\_conversation/1 | Win32: Close DDE channel | Win32: Закрыть канал DDE |
| close\_shared\_object/1 | UNIX: Close shared library (.so file) | UNIX: Закрыть разделяемую библиотеку (.so файл) |
| collation\_key/2 | Sort key for locale dependent ordering | ключ сортировки для Локально зависимое упорядочения |
| comment\_hook/3 | (hook) handle comments in sources | (хук) обрабатывать комментарии в источниках |
| compare/3 | Compare, using a predicate to determine the order | Сравнить, используя предикат для определения порядка |
| compile\_aux\_clauses/1 | Compile predicates for goal\_expansion/2 | Компилировать предикаты для goal\_expansion/2 |
| compile\_predicates/1 | Compile dynamic code to static | Компиляция динамический код статике |
| compiling/0 | Is this a compilation run? | Является ли это компиляция запущеной? |
| compound/1 | Test for compound term | Испытание на составное утверждение |
| compound\_name\_arity/3 | Name and arity of a compound term | Имя и Арность утверждения соединения |
| compound\_name\_arguments/3 | Name and arguments of a compound term | Имя и аргументы утверждения соединения |
| code\_type/2 | Classify a character-code | Классифицировать Символьный код |
| consult/1 | Read (compile) a Prolog source file | Read (компилировать) исходный файл Пролог |
| context\_module/1 | Get context module of current goal | Получить контекст модуль текущей цели |
| convert\_time/8 | Break time stamp into fields | Перерыв отметки времени в полях |
| convert\_time/2 | Convert time stamp to string | Преобразовать временную метку в строку |
| copy\_stream\_data/2 | Copy all data from stream to stream | Копировать все данные из потока в поток |
| copy\_stream\_data/3 | Copy n bytes from stream to stream | Копирование N байтов из потока в поток |
| copy\_predicate\_clauses/2 | Copy clauses between predicates | Копирование статей между предикаты |
| copy\_term/2 | Make a copy of a term | Сделать копию утверждение |
| copy\_term/3 | Copy a term and obtain attribute-goals | Скопируйте утверждение и получить атрибут-целей |
| copy\_term\_nat/2 | Make a copy of a term without attributes | Сделайте копию утверждение без атрибутов |
| create\_prolog\_flag/3 | Create a new Prolog flag | Создайте новый флаг пролога |
| current\_arithmetic\_function/1 | Examine evaluable functions | Рассмотрение оцениваемые функции |
| current\_atom/1 | Examine existing atoms | Рассмотрение существующих атомов |
| current\_blob/2 | Examine typed blobs | Рассмотрение напечатанные сгустки |
| current\_char\_conversion/2 | Query input character mapping | Отображение ввода символов запроса |
| current\_flag/1 | Examine existing flags | Рассмотрение существующих флагов |
| current\_foreign\_library/2 | library(shlib) Examine loaded shared libraries (.so files) | библиотека (shlib) Рассмотрение загруженных разделяемых библиотек (файлы .so) |
| current\_format\_predicate/2 | Enumerate user-defined format codes | Перечислять определяемые пользователем коды формата |
| current\_functor/2 | Examine existing name/arity pairs | Рассмотрение существующих пар имя / арность |
| current\_input/1 | Get current input stream | Получить текущий входной поток |
| current\_key/1 | Examine existing database keys | Рассмотрение существующих ключей базы данных |
| current\_locale/1 | Get the current locale | Получить текущий языковой стандарт |
| current\_module/1 | Examine existing modules | Рассмотрение существующих модулей |
| current\_op/3 | Examine current operator declarations | Рассмотрение текущего оператора объявления |
| current\_output/1 | Get the current output stream | Получить текущий выходной поток |
| current\_predicate/1 | Examine existing predicates (ISO) | Рассмотрение существующих предикатов (ISO) |
| current\_predicate/2 | Examine existing predicates | Рассмотрение существующих предикатов (ISO) |
| current\_signal/3 | Current software signal mapping | Текущее сопоставление программного сигнала |
| current\_stream/3 | Examine open streams | Рассмотрение открытых потоков |
| cyclic\_term/1 | Test term for cycles | утверждение испытаний для циклов |
| day\_of\_the\_week/2 | Determine ordinal-day from date | Извлечь день из даты |
| date\_time\_stamp/2 | Convert date structure to time-stamp | Преобразование структуру даты в формат времени |
| date\_time\_value/3 | Extract info from a date structure | Извлечение информации из структуры даты |
| dcg\_translate\_rule/2 | Source translation of DCG rules | Источник перевод правил DCG |
| dcg\_translate\_rule/4 | Source translation of DCG rules | Источник перевод правил DCG |
| dde\_current\_connection/2 | Win32: Examine open DDE connections | Win32: Рассмотрение открытых соединений DDE |
| dde\_current\_service/2 | Win32: Examine DDE services provided | Win32: Рассмотрение сервисов, предоставляемх DDE |
| dde\_execute/2 | Win32: Execute command on DDE server | Win32: Выполнить команду на сервере DDE |
| dde\_register\_service/2 | Win32: Become a DDE server | Win32: Стать сервером DDE |
| dde\_request/3 | Win32: Make a DDE request | Win32: Сделать запрос DDE |
| dde\_poke/3 | Win32: POKE operation on DDE server | Win32: POKE операция DDE сервера |
| dde\_unregister\_service/1 | Win32: Terminate a DDE service | Win32: Завершить сервис DDE |
| debug/0 | Test for debugging mode | Проверить режим отладки |
| debug/1 | Select topic for debugging | Выбрать тему для отладки |
| debug/3 | Print debugging message on topic | Печать сообщение отладки в тему |
| debug\_control\_hook/1 | (hook) Extend spy/1, etc. | (хук) Расширить шпиона/1, и т.д. |
| debugging/0 | Show debugger status | Показать статус отладчика |
| debugging/1 | Test where we are debugging topic | Определить где мы в теме отладки |
| default\_module/2 | Query module inheritance | Запросить наследование модуля |
| del\_attr/2 | Delete attribute from variable | Удалить атрибут из переменной |
| del\_attrs/1 | Delete all attributes from variable | Удалить все атрибуты из переменной |
| del\_dict/4 | Delete Key-Value pair from a dict | Удалить пару "ключ-значение" из справочник |
| delete\_directory/1 | Remove a folder from the file system | Удалить папку из файловой системы |
| delete\_file/1 | Remove a file from the file system | Удалить файл из файловой системы |
| delete\_import\_module/2 | Remove module from import list | Удалить модуль из списка импорта |
| deterministic/1 | Test deterministicy of current clause | Тест deterministicy текущего пункта |
| dif/2 | Constrain two terms to be different | Задать различие утверждений |
| directory\_files/2 | Get entries of a directory/folder | Получить данные из каталога / папки |
| discontiguous/1 | Indicate distributed definition of a predicate | Указывает определение распределенного предиката |
| divmod/4 | Compute quotient and remainder of two integers | Вычисляет частное и остаток двух целых чисел |
| downcase\_atom/2 | Convert atom to lower-case | Преобразовать атом в нижний регистр |
| duplicate\_term/2 | Create a copy of a term | Создайть копию утверждений |
| dwim\_match/2 | Atoms match in ``Do What I Mean'' sense | Атомы в смысле `` Делай то, что я имею ввиду '' |
| dwim\_match/3 | Atoms match in ``Do What I Mean'' sense | Атомы в смысле `` Делай то, что я имею ввиду '' |
| dwim\_predicate/2 | Find predicate in ``Do What I Mean'' sense | Найти предикат в `` Do What I Mean '' смысле |
| dynamic/1 | Indicate predicate definition may change | Показывают, что определение предиката может изменяться |
| edit/0 | Edit current script- or associated file | Редактировать текущий сценарий или связанный с ним файл |
| edit/1 | Edit a file, predicate, module (extensible) | Редактировать файл, предикат, модуль (расширяемый) |
| elif/1 | Part of conditional compilation (directive) | Часть условной компиляции (директива) |
| else/0 | Part of conditional compilation (directive) | Часть условной компиляции (директива) |
| empty\_assoc/1 | Create/test empty association tree | Создать / протестировать пустое дерево связей |
| empty\_nb\_set/1 | Test/create an empty не-backtrackable set | Протестировать/создать пустой не backtrackable набор |
| encoding/1 | Define encoding inside a source file | Определить кодировку внутри исходного файла |
| endif/0 | End of conditional compilation (directive) | Конец условной компиляции (директива) |
| ensure\_loaded/1 | Consult a file if that has not yet been done | Обратиться к файлу, если он еще не готов |
| erase/1 | Erase a database record or clause | Стереть запись базы данных или пункт |
| eval\_license/0 | Evaluate licenses of loaded modules | Оценка лицензий загруженных модулей |
| exception/3 | (hook) Handle runtime exceptions | (хук) Обработки исключений среды выполнения |
| exists\_directory/1 | Check existence of directory | Проверка существования каталога |
| exists\_file/1 | Check existence of file | Проверить существование файла |
| exists\_source/1 | Check existence of a Prolog source | Проверить существование источника пролога |
| expand\_answer/2 | Expand answer of query | Расширить ответ на запроса |
| expand\_file\_name/2 | Wildcard expansion of file names | Wildcard расширение имен файлов |
| expand\_file\_search\_path/2 | Wildcard expansion of file paths | Wildcard расширение путей к файлам |
| expand\_goal/2 | Compiler: expand goal in clause-body | Компилятор: расширить цели в пункте тела |
| expand\_goal/4 | Compiler: expand goal in clause-body | Компилятор: расширить цели в пункте тела |
| expand\_query/4 | Expanded entered query | Расширенное введение запроса |
| expand\_term/2 | Compiler: expand read term into clause(s) | Компилятор: расширеное чтение утверждения в пункте (с) |
| expand\_term/4 | Compiler: expand read term into clause(s) | Компилятор: расширеное чтение утверждения в пункте (с) |
| expects\_dialect/1 | For which Prolog dialect is this code written? | Для кокого диалекта Пролога написан этот код? |
| explain/1 | library(explain) Explain argument | библиотека (объяснение) Объяснить аргумент |
| explain/2 | library(explain) 2nd argument is explanation of first | библиотека (объясниние) 2-й аргумент объясняющий первой |
| export/1 | Export a predicate from a module | Экспорт предиката из модуля |
| fail/0 | Always false | Всегда ложь |
| false/0 | Always false | Всегда ложь |
| current\_prolog\_flag/2 | Get system configuration parameters | Получить параметры конфигурации системы |
| file\_base\_name/2 | Get file part of path | Получить часть пути, относящуюся к файлу |
| file\_directory\_name/2 | Get directory part of path | Получить часть пути, относящуюся к каталогу |
| file\_name\_extension/3 | Add, remove or test file extensions | Добавить, удалить или протестировать расширения файлов |
| file\_search\_path/2 | Define path-aliases for locating files | Определить псевдонимы пути для поиска файлов |
| find\_chr\_constraint/1 | Returns a constraint from the store | Возвращает ограничение из хранилища |
| findall/3 | Find all solutions to a goal | Найти все решения цели |
| findall/4 | Difference list version of findall/3 | Разница версий списков FindAll / 3 |
| findnsols/4 | Find first N solutions | Найти первые N решений |
| findnsols/5 | Difference list version of findsols/4 | Разница версий списков findsols / 4 |
| flag/3 | Simple global variable system | Простая система глобальной переменной |
| float/1 | Type check for a floating point number | Проверка типа для числа с плавающей запятой |
| flush\_output/0 | Output pending characters on current stream | Ожидание вывода символов в текущий поток |
| flush\_output/1 | Output pending characters on specified stream | Ожидание вывода символов в указанный поток |
| forall/2 | Prove goal for all solutions of another goal | Доказать цель для всех решения другой цели |
| format/1 | Formatted output | Форматированный вывод |
| format/2 | Formatted output with arguments | Форматированный вывод с аргументами |
| format/3 | Formatted output on a stream | Форматированный вывод в поток |
| format\_time/3 | C strftime() like date/time formatter | C STRFTIME (), похожий формат дата / время |
| format\_time/4 | date/time formatter with explicit locale | Формат даты / времени с явным определением |
| format\_predicate/2 | Program format/[1,2] | Формат программы / [1,2] |
| term\_attvars/2 | Find attributed variables in a term | Найти приписываемые переменные в утверждении |
| term\_variables/2 | Find unbound variables in a term | Найти несвязанных переменные в утверждении |
| term\_variables/3 | Find unbound variables in a term | Найти несвязанных переменные в утверждении |
| text\_to\_string/2 | Convert arbitrary text to a string | Преобразование произвольный текст в строку |
| freeze/2 | Delay execution until variable is bound | Задержать выполнение до тех пор, пока переменная привязана |
| frozen/2 | Query delayed goals on var | Задержка запроса цели на var |
| functor/3 | Get name and arity of a term or construct a term | Получить имя и арность утверждения или построить утверждение |
| garbage\_collect/0 | Invoke the garbage collector | Вызов сборщика мусора |
| garbage\_collect\_atoms/0 | Invoke the atom garbage collector | Вызов сборщика мусора атома |
| garbage\_collect\_clauses/0 | Invoke clause garbage collector | Вызов сборщика мусора утверждения |
| gen\_assoc/3 | Enumerate members of association tree | Перечислить членов дерева ассоциации |
| gen\_nb\_set/2 | Generate members of не-backtrackable set | Сформировать членов не backtrackable набора |
| gensym/2 | Generate unique atoms from a base | Создание уникальных атомов из базы |
| get/1 | Read first non-blank character | Прочитать первый непустой символ |
| get/2 | Read first non-blank character from a stream | Прочитать первый непустой символ из потока |
| get\_assoc/3 | Fetch key from association tree | Извлечь ключ из дерева ассоциации |
| get\_assoc/5 | Fetch key from association tree | Извлечь ключ из дерева ассоциации |
| get\_attr/3 | Fetch named attribute from a variable | Выбрать именованный атрибут из переменной |
| get\_attrs/2 | Fetch all attributes of a variable | Получить все атрибуты переменной |
| get\_byte/1 | Read next byte (ISO) | Прочитать следующий байт (ISO) |
| get\_byte/2 | Read next byte from a stream (ISO) | Прочитать следующий байт из потока (ISO) |
| get\_char/1 | Read next character as an atom (ISO) | Прочитать следующий символ как atom (ISO) |
| get\_char/2 | Read next character from a stream (ISO) | Прочитать следующий символ из потока (ISO) |
| get\_code/1 | Read next character (ISO) | Прочитать следующий символ (ISO) |
| get\_code/2 | Read next character from a stream (ISO) | Прочитать следующий символ из потока (ISO) |
| get\_dict/3 | Get the value associated to a key from a dict | Получить значение, связанное с ключом из словаря |
| get\_dict/5 | Replace existing value in a dict | Заменить существующее значение в словаре |
| get\_flag/2 | Get value of a flag | Получить значение флага |
| get\_single\_char/1 | Read next character from the terminal | Прочитать следующий символ из терминала |
| get\_string\_code/3 | Get character code at index in string | Получить код символа по индексу в строке |
| get\_time/1 | Get current time | Получить текущее время |
| get0/1 | Read next character | Прочитать следующий символ |
| get0/2 | Read next character from a stream | Прочитать следующий символ из потока |
| getenv/2 | Get shell environment variable | Получить переменную среды оболочки |
| goal\_expansion/2 | Hook for macro-expanding goals | Хук для макро расширения целей |
| goal\_expansion/4 | Hook for macro-expanding goals | Хук для макро расширения целей |
| ground/1 | Verify term holds no unbound variables | Убедитесь, что термин содержит не свободные переменные |
| gdebug/0 | Debug using graphical tracer | Отладка с помощью графического трассирующика |
| gspy/1 | Spy using graphical tracer | Шпион, с использованием графического трассирующика |
| gtrace/0 | Trace using graphical tracer | Трассировка с помощью графического трассирующика |
| guitracer/0 | Install hooks for the graphical debugger | Установить хук для графического отладчика |
| gxref/0 | Cross-reference loaded program | Программа, загружаемая по перекрестной ссылке |
| halt/0 | Exit from Prolog | Выход из Пролога |
| halt/1 | Exit from Prolog with status | Выход из Пролога со статусом |
| term\_hash/2 | Hash-value of ground term | Хэш-значение утверждения |
| term\_hash/4 | Hash-value of term with depth limit | Хэш-значение утверждения с ограничением глубины |
| help/0 | Give help on help | Дать справку о помощи |
| help/1 | Give help on predicates and show parts of manual | Дать справку по предикатам и показать часть инструкции |
| help\_hook/1 | (hook) User-hook in the help-system | (Хук) Пользовательский Хук в системе справки |
| if/1 | Start conditional compilation (directive) | Начало условной компиляции (директива) |
| ignore/1 | Call the argument, but always succeed | Вызов аргумент, но всегда упешный |
| import/1 | Import a predicate from a module | Импортировать предиката из модуля |
| import\_module/2 | Query import modules | Запрос импорта модулей |
| in\_pce\_thread/1 | Run goal in XPCE thread | Выполнить цель в XPCE потоке |
| in\_pce\_thread\_sync/1 | Run goal in XPCE thread | Выполнить цель в XPCE потоке |
| include/1 | Include a file with declarations | Включить файл с объявлениями |
| initialization/1 | Initialization directive | директива инициализации |
| initialization/2 | Initialization directive | директива инициализации |
| instance/2 | Fetch clause or record from reference | Извлечь статью или запись из ссылки |
| integer/1 | Type check for integer | Проверка типа целого числа (integer) |
| interactor/0 | Start new thread with console and top level | Начать новый поток с консолью и верхним уровнем |
| is/2 | Evaluate arithmetic expression | Оценить арифметическое выражение |
| is\_absolute\_file\_name/1 | True if arg defines an absolute path | Правда, если аргумент определяет абсолютный путь |
| is\_assoc/1 | Verify association list | Проверка списка ассоциаций |
| is\_list/1 | Type check for a list | Проверка типа для списка |
| is\_dict/1 | Type check for a dict | Проверка типа для справочник |
| is\_dict/2 | Type check for a dict in a class | Проверка типа для справочник в классе |
| is\_stream/1 | Type check for a stream handle | Проверка типа для дескриптора потока |
| join\_threads/0 | Join all terminated threads interactively | Соединение всех прекращенных потоков интерактивно |
| keysort/2 | Sort, using a key | Сортировка с помощью ключа |
| last/2 | Last element of a list | Последний элемент списка |
| leash/1 | Change ports visited by the tracer | Изменить порты опрашиваемые трейсером |
| length/2 | Length of a list | Длина списка |
| library\_directory/1 | (hook) Directories holding Prolog libraries | (Хук) Пути с библиотеками Prolog |
| license/1 | Define license for current file | Определить лицензию для текущего файла |
| license/2 | Define license for named module | Определить лицензию для именованных модуля |
| line\_count/2 | Line number on stream | Номер строки в потоке |
| line\_position/2 | Character position in line on stream | Позиция символа в строке в потоке |
| list\_debug\_topics/0 | List registered topics for debugging | Список зарегистрированных тем для отладки |
| list\_to\_assoc/2 | Create association tree from list | Создание дерева ассоциаций из списка |
| list\_to\_set/2 | Remove duplicates from a list | Удалить дубликаты из списка |
| list\_strings/0 | Help porting to version 7 | Помощь в портировании на 7-ую версию |
| listing/0 | List program in current module | Список программ в текущем модуле |
| listing/1 | List predicate | Список предикат |
| load\_files/1 | Load source files | Загружать исходные файлы |
| load\_files/2 | Load source files with options | Загружать исходные файлы с параметрами |
| load\_foreign\_library/1 | library(shlib) Load shared library (.so file) | библиотека (shlib) Загрузить разделяемую библиотеку (.so файл) |
| load\_foreign\_library/2 | library(shlib) Load shared library (.so file) | библиотека (shlib) Загрузить разделяемую библиотеку (.so файл) |
| locale\_create/3 | Create a new locale object | Создание нового объекта языкового стандарта |
| locale\_destroy/1 | Destroy a locale object | Уничтожить объект языкового стандарта |
| locale\_property/2 | Query properties of locale objects | Свойства запроса объектов языкового стандарта |
| locale\_sort/2 | Language dependent sort of atoms | Язык зависит от рода атомов |
| make/0 | Reconsult all changed source files | Проконсультировать все изменения исходных файлов |
| make\_directory/1 | Create a folder on the file system | Создайте папку в файловой системе |
| make\_library\_index/1 | Create autoload file INDEX.pl | Создать файл автозагрузки INDEX.pl |
| make\_library\_index/2 | Create selective autoload file INDEX.pl | Создание селективного файла автозагрузки INDEX.pl |
| map\_assoc/2 | Map association tree | Карта дерева ассоциаций |
| map\_assoc/3 | Map association tree | Карта дерева ассоциаций |
| dict\_create/3 | Create a dict from data | Создание справочника из данных |
| dict\_pairs/3 | Convert between dict and list of pairs | Преобразование между справочник и списком пар |
| max\_assoc/3 | Highest key in association tree | Самая высокая ключевая роль в дереве ассоциаций |
| memberchk/2 | Deterministic member/2 | Детерминированный член/2 |
| message\_hook/3 | Intercept print\_message/2 | Перехват print\_message/2 |
| message\_line\_element/2 | (hook) Intercept print\_message\_lines/3 | (хук) Перехват print\_message\_lines / 3 |
| message\_property/2 | (hook) Define display of a message | (Хук) Определение отображения сообщения |
| message\_queue\_create/1 | Create queue for thread communication | Создание очереди для потоков коммуникации |
| message\_queue\_create/2 | Create queue for thread communication | Создание очереди для потоков коммуникации |
| message\_queue\_destroy/1 | Destroy queue for thread communication | Уничтожить очереди для потоков коммуникации |
| message\_queue\_property/2 | Query message queue properties | Запрос свойств очереди сообщений |
| message\_to\_string/2 | Translate message-term to string | Перевести сообщение-утверждение в строку |
| meta\_predicate/1 | Declare access to other predicates | Объявить доступ к другим предикатам |
| min\_assoc/3 | Lowest key in association tree | Самая низкая ключевая роль в дереве ассоциаций |
| module/1 | Query/set current type-in module | Запрос и установка текущего типа в модуле |
| module/2 | Declare a module | Объявить модуль |
| module/3 | Declare a module with language options | Объявить модуль с параметрами языка |
| module\_property/2 | Find properties of a module | Найти свойства модуля |
| module\_transparent/1 | Indicate module based meta-predicate | Указать модуль на основе мета предиката |
| msort/2 | Sort, do not remove duplicates | Сортировка без удаления дубликатов |
| multifile/1 | Indicate distributed definition of predicate | Указать определение распределенного предиката |
| mutex\_create/1 | Create a thread-synchronisation device | Создание устройства синхронизации потоков |
| mutex\_create/2 | Create a thread-synchronisation device | Создание устройства синхронизации потоков |
| mutex\_destroy/1 | Destroy a mutex | Уничтожить мьютекс |
| mutex\_lock/1 | Become owner of a mutex | Стать владельцем мьютекса |
| mutex\_property/2 | Query mutex properties | Свойства запроса мьютекс |
| mutex\_statistics/0 | Print statistics on mutex usage | Печать статистики по использованию мьютекс |
| mutex\_trylock/1 | Become owner of a mutex (non-blocking) | Стать владельцем мьютекса (без блокировки) |
| mutex\_unlock/1 | Release ownership of mutex | Освободить владение мьютексом |
| mutex\_unlock\_all/0 | Release ownership of all mutexes | Освободить владение всеми мьютексами |
| name/2 | Convert between atom and list of character codes | Преобразование между атомом и списком кодов символов |
| nb\_current/2 | Enumerate не-backtrackable global variables | Перечислять не-backtrackable глобальные переменные |
| nb\_delete/1 | Delete a не-backtrackable global variable | Удаление не-backtrackable глобальной переменной |
| nb\_getval/2 | Fetch не-backtrackable global variable | Извлечь не-backtrackable глобальной переменной |
| nb\_link\_dict/3 | не-backtrackable assignment to dict | не-backtrackable присвоение списку |
| nb\_linkarg/3 | не-backtrackable assignment to term | не-backtrackable присвоение утверждению |
| nb\_linkval/2 | Assign не-backtrackable global variable | Назначить не-backtrackable глобальной переменной |
| nb\_set\_to\_list/2 | Convert не-backtrackable set to list | Преобразование не-backtrackable набор в список |
| nb\_set\_dict/3 | не-backtrackable assignment to dict | не-backtrackable присвоение списка |
| nb\_setarg/3 | не-backtrackable assignment to term | не-backtrackable присвоение утверждения |
| nb\_setval/2 | Assign не-backtrackable global variable | Назначить не-backtrackable глобальной переменной |
| nl/0 | Generate a newline | Генерация новой строки |
| nl/1 | Generate a newline on a stream | Генерация новой строки на поток |
| nodebug/0 | Disable debugging | Отключить отладку |
| nodebug/1 | Disable debug-topic | Отключить отладку темы |
| noguitracer/0 | Disable the graphical debugger | Отключить графический отладчик |
| nonvar/1 | Type check for bound term | Проверка типа для связанного утверждения |
| noprofile/1 | Hide (meta-) predicate for the profiler | Скрыть (мета-) предикат для профилировщика |
| noprotocol/0 | Disable logging of user interaction | Отключить ведение журнала взаимодействия с пользователем |
| normalize\_space/2 | Normalize white space | Нормализовать белое пространство |
| nospy/1 | Remove spy point | Удалить шпионскую точку |
| nospyall/0 | Remove all spy points | Удалить все точки шпиона |
| not/1 | Negation by failure (argument not provable). Same as \+/1 | Отрицание отказом (аргумент не доказуем). То же, что \ + / 1 |
| notrace/0 | Stop tracing | Остановить трассировку |
| notrace/1 | Do not debug argument goal | Не отлаживать аргумент цели |
| nth\_clause/3 | N-th clause of a predicate | N-ое предложение предиката |
| nth\_integer\_root\_and\_remainder/4 | Integer root and remainder | Целочисленное корень и остаток |
| number/1 | Type check for integer or float | Проверка типа для целого числа или числа с плавающей точкой |
| number\_chars/2 | Convert between number and one-char atoms | Преобразование между числом и односимвольным атомом |
| number\_codes/2 | Convert between number and character codes | Преобразование между числом и символьным кодом |
| number\_string/2 | Convert between number and string | Преобразование между числом и строкой |
| numbervars/3 | Number unbound variables of a term | Число свободных переменных утверждения |
| numbervars/4 | Number unbound variables of a term | Число свободных переменных утверждения |
| on\_signal/3 | Handle a software signal | Обработка сигнала программного обеспечения |
| once/1 | Call a goal deterministically | Детерминированный вызов целей |
| op/3 | Declare an operator | Объявление оператора |
| open/3 | Open a file (creating a stream) | Открыть файл (создание потока) |
| open/4 | Open a file (creating a stream) | Открыть файл (создание потока) |
| open\_dde\_conversation/3 | Win32: Open DDE channel | Win32: Открыть канал DDE |
| open\_null\_stream/1 | Open a stream to discard output | Открыть поток для отмены вывода |
| open\_resource/3 | Open a program resource as a stream | Открыть ресурс программы как поток |
| open\_shared\_object/2 | UNIX: Open shared library (.so file) | UNIX: Открыть разделяемую библиотеку (.so-файл) |
| open\_shared\_object/3 | UNIX: Open shared library (.so file) | UNIX: Открыть разделяемую библиотеку (.so-файл) |
| open\_string/2 | Open a string as a stream | Открыть строку в виде потока |
| ord\_list\_to\_assoc/2 | Convert ordered list to assoc | Преобразование упорядоченный список в ассоциативный |
| parse\_time/2 | Parse text to a time-stamp | Анализировать текст на наличие времени |
| parse\_time/3 | Parse text to a time-stamp | Анализировать текст на наличие времени |
| pce\_dispatch/1 | Run XPCE GUI in separate thread | Запустить XPCE GUI в отдельном потоке |
| pce\_call/1 | Run goal in XPCE GUI thread | Выполнить цель в XPCE GUI потоке |
| peek\_byte/1 | Read byte without removing | Прочитать байт без удаления |
| peek\_byte/2 | Read byte without removing | Прочитать байт без удаления |
| peek\_char/1 | Read character without removing | Чтение символов без удаления |
| peek\_char/2 | Read character without removing | Чтение символов без удаления |
| peek\_code/1 | Read character-code without removing | Читать код символа без удаления |
| peek\_code/2 | Read character-code without removing | Читать код символа без удаления |
| peek\_string/3 | Read a string without removing | Прочитать строку без удаления |
| phrase/2 | Activate grammar-rule set | Активировать набор грамматических правил |
| phrase/3 | Activate grammar-rule set (returning rest) | Активировать набор грамматических правил (с возвратом параметра) |
| phrase\_from\_quasi\_quotation/2 | Parse quasi quotation with DCG | Разбираем квази котировку с DCG |
| please/3 | Query/change environment parameters | Запросить/изменить параметры окружающей среды |
| plus/3 | Logical integer addition | Логическое дополнение целых чисел |
| portray/1 | (hook) Modify behaviour of print/1 | (хук) Изменить поведение печати / 1 |
| portray\_clause/1 | Pretty print a clause | Достаточно распечатать предложение |
| portray\_clause/2 | Pretty print a clause to a stream | Достаточно распечатать предложение в поток |
| predicate\_property/2 | Query predicate attributes | Запрос атрибутов предиката |
| predsort/3 | Sort, using a predicate to determine the order | Сортировка используя предикат для определения порядка |
| print/1 | Print a term | Печать утверждений |
| print/2 | Print a term on a stream | Печать утверждений в поток |
| print\_message/2 | Print message from (exception) term | Распечатать сообщение от (исключение) утверждения |
| print\_message\_lines/3 | Print message to stream | Печатать сообщения в поток |
| profile/1 | Obtain execution statistics | Получить статистику выполнения |
| profile/2 | Obtain execution statistics | Получить статистику выполнения |
| profile\_count/3 | Obtain profile results on a predicate | Получить профильные результаты по предикату |
| profiler/2 | Obtain/change status of the profiler | Получите статус / изменение профилировщика |
| prolog/0 | Run interactive top level | Запуск интерактивного верхнего уровня |
| prolog\_choice\_attribute/3 | Examine the choice point stack | Осмотрите стек точки выбора |
| prolog\_current\_choice/1 | Reference to most recent choice point | Ссылка на самой последней точки выбора |
| prolog\_current\_frame/1 | Reference to goal's environment stack | Ссылка на стек среды цель в |
| prolog\_cut\_to/1 | Realise global cuts | Осознайте глобальных сокращений |
| prolog\_edit:locate/2 | Locate targets for edit/1 | Найдите цели для редактирования / 1 |
| prolog\_edit:locate/3 | Locate targets for edit/1 | Найдите цели для редактирования / 1 |
| prolog\_edit:edit\_source/1 | Call editor for edit/1 | Вызов редактор для редактирования / 1 |
| prolog\_edit:edit\_command/2 | Specify editor activation | Указать активацию редактора |
| prolog\_edit:load/0 | Load edit/1 extensions | Загрузка расширений редактирования/1 |
| prolog\_exception\_hook/4 | Rewrite exceptions | Переписать исключения |
| prolog\_file\_type/2 | Define meaning of file extension | Определить значение расширения файла |
| prolog\_frame\_attribute/3 | Obtain information on a goal environment | Получить информацию о среде цели |
| prolog\_ide/1 | Program access to the development environment | Программный доступ к среде разработки |
| prolog\_list\_goal/1 | (hook) Intercept tracer 'L' command | (Хук) Перехват трейсера «L» команд |
| prolog\_load\_context/2 | Context information for directives | Сведения о контексте для директив |
| prolog\_load\_file/2 | (hook) Program load\_files/2 | (хук) Программа загрузки файлов / 2 |
| prolog\_skip\_level/2 | Indicate deepest recursion to trace | Указать самую глубокую рекурсию для трассировки |
| prolog\_skip\_frame/1 | Perform `skip' on a frame | Выполнить `пропустить 'на кадре (фрейме) |
| prolog\_stack\_property/2 | Query properties of the stacks | Свойства запроса стеков |
| prolog\_to\_os\_filename/2 | Convert between Prolog and OS filenames | Преобразование имен файлов между Прологом и операционной системой |
| prolog\_trace\_interception/4 | library(user) Intercept the Prolog tracer | Библиотека (пользователя) Перехват трейсера Пролог |
| project\_attributes/2 | Project constraints to query variables | Ограничения проекта для запроса переменные |
| prompt1/1 | Change prompt for 1 line | Изменить запрос на 1 линии |
| prompt/2 | Change the prompt used by read/1 | Измените строку, используемую для чтения / 1 |
| protocol/1 | Make a log of the user interaction | Сделать журнал взаимодействия пользователя |
| protocola/1 | Append log of the user interaction to file | Append журнал взаимодействия пользователя в файл |
| protocolling/1 | On what file is user interaction logged | На какой файл является взаимодействие пользователя, вошедшего |
| public/1 | Declaration that a predicate may be called | Декларация о том, что предикат может быть назван |
| put/1 | Write a character | Записать символ |
| put/2 | Write a character on a stream | Записать символ на поток |
| put\_assoc/4 | Add Key-Value to association tree | Добавить ключевое значение для дерева ассоциаций |
| put\_attr/3 | Put attribute on a variable | Установить атрибут переменной |
| put\_attrs/2 | Set/replace all attributes on a variable | Установить / заменить все атрибуты переменной |
| put\_byte/1 | Write a byte | Записать байт |
| put\_byte/2 | Write a byte on a stream | Записать байт в поток |
| put\_char/1 | Write a character | Записать символ |
| put\_char/2 | Write a character on a stream | Записать символ в поток |
| put\_code/1 | Write a character-code | Записать код символа |
| put\_code/2 | Write a character-code on a stream | Записать код символа в поток |
| put\_dict/3 | Add/replace multiple keys in a dict | Добавить / заменить несколько ключей в Словаре |
| put\_dict/4 | Add/replace a single key in a dict | Добавить / заменить один ключ в Словаре |
| qcompile/1 | Compile source to Quick Load File | Скомпилировать исходник быстрой загрузки файла |
| qcompile/2 | Compile source to Quick Load File | Скомпилировать исходник быстрой загрузки файла |
| qsave\_program/1 | Create runtime application | Создание приложения среды выполнения |
| qsave\_program/2 | Create runtime application | Создание приложения среды выполнения |
| quasi\_quotation\_syntax/1 | Declare quasi quotation syntax | Объявить квази синтаксис цитаты |
| quasi\_quotation\_syntax\_error/1 | Raise syntax error | Вызвать ошибку синтаксиса |
| random\_property/1 | Query properties of random generation | Запрос свойства случайной генерации |
| rational/1 | Type check for a rational number | Проверка типа рационального числа |
| rational/3 | Decompose a rational | Разложить рациональное |
| read/1 | Read Prolog term | Прочитать утверждение Пролог |
| read/2 | Read Prolog term from stream | Прочитать утверждение Пролог из потока |
| read\_clause/3 | Read clause from stream | Прочитать статью из потока |
| read\_history/6 | Read using history substitution | Прочитать используя историю замещения |
| read\_link/3 | Read a symbolic link | Прочитать символическую ссылку |
| read\_pending\_input/3 | Fetch buffered input from a stream | Извлечь буферизованный ввод из потока |
| read\_string/3 | Read a number of characters into a string | Прочитать числа символов в строку |
| read\_string/5 | Read string upto a delimiter | ПроПрочитатьстроки по разделителю |
| read\_term/2 | Read term with options | Прочитать утверждение с параметрами |
| read\_term/3 | Read term with options from stream | Прочитать утверждение с параметрами из потока |
| read\_term\_from\_atom/3 | Read term with options from atom | Прочитать утверждение с параметрами из атома |
| recorda/2 | Record term in the database (first) | Записать утверждение в базу данных (первый) |
| recorda/3 | Record term in the database (first) | Записать утверждение в базу данных (первый) |
| recorded/2 | Obtain term from the database | Получить утверждение из базы данных |
| recorded/3 | Obtain term from the database | Получить утверждение из базы данных |
| recordz/2 | Record term in the database (last) | Записать утверждение в базу данных (последний) |
| recordz/3 | Record term in the database (last) | Записать утверждение в базу данных (последний) |
| redefine\_system\_predicate/1 | Abolish system definition | Упразднить определение системы |
| reexport/1 | Load files and re-export the imported predicates | Загрузка файлов и реэкспорт импортированных предикатов |
| reexport/2 | Load predicates from a file and re-export it | Загрузка предикатов из файла и реэкспорта |
| reload\_foreign\_libraries/0 | Reload DLLs/shared objects | Перезагрузка библиотеки DLL/общих объектов |
| reload\_library\_index/0 | Force reloading the autoload index | Принудительная перезагрузка индекса автозагрузки |
| rename\_file/2 | Change name of file | Изменить имя файла |
| repeat/0 | Succeed, leaving infinite backtrack points | Успешно завершить, оставляя бесконечные точки BACKTRACK |
| require/1 | This file requires these predicates | Этот файл требует данных предикатов |
| reset\_gensym/1 | Reset a gensym key | Сброс ключа gensym |
| reset\_gensym/0 | Reset all gensym keys | Сброс всех ключей gensym |
| reset\_profiler/0 | Clear statistics obtained by the profiler | Очистить статистические данные, полученные профилировщиком |
| resource/3 | Declare a program resource | Объявить ресурс программы |
| retract/1 | Remove clause from the database | Удалить статью из базы данных |
| retractall/1 | Remove unifying clauses from the database | Удалить унифицированнве положения из базы данных |
| same\_file/2 | Succeeds if arguments refer to same file | Завершится успешно если аргументы относятся к тому же файлу |
| same\_term/2 | Test terms to be at the same address | Проверка утверждений на нахождение по одному адресу |
| see/1 | Change the current input stream | Изменение текущего входного потока |
| seeing/1 | Query the current input stream | Запрос текущего входного потока |
| seek/4 | Modify the current position in a stream | Изменить текущую позицию в потоке |
| seen/0 | Close the current input stream | Закрытие текущего входного потока |
| select\_dict/2 | Select matching attributes from a dict | Выбор соответствия атрибутов из справочника |
| select\_dict/3 | Select matching attributes from a dict | Выбор соответствия атрибутов из справочника |
| set\_end\_of\_stream/1 | Set physical end of an open file | Установить физический конец открытого файла |
| set\_flag/2 | Set value of a flag | Установленное значение флага |
| set\_input/1 | Set current input stream from a stream | Установка текущего входного потока из потока |
| set\_locale/1 | Set the default local | Установить по умолчанию локальный |
| set\_module/1 | Set properties of a module | Установить свойства модуля |
| set\_output/1 | Set current output stream from a stream | Установить текущий выходной поток из потока |
| set\_prolog\_IO/3 | Prepare streams for interactive session | Подготовка потоков для интерактивного сеанса |
| set\_prolog\_flag/2 | Define a system feature | Определение функции системы |
| set\_prolog\_stack/2 | Modify stack characteristics | Изменение характеристик стека |
| set\_random/1 | Control random number generation | Контроль генерации случайных чисел |
| set\_stream/2 | Set stream attribute | Установить атрибут потока |
| set\_stream\_position/2 | Seek stream to position | Поиск позиции в потоке |
| setup\_call\_cleanup/3 | Undo side-effects safely | Безопасно отменить побочные эффекты |
| setup\_call\_catcher\_cleanup/4 | Undo side-effects safely | Безопасно отменить побочные эффекты |
| setarg/3 | Destructive assignment on term | Назначить утверждение на разрушение |
| setenv/2 | Set shell environment variable | Установить переменнаую среды набора оболочки |
| setlocale/3 | Set/query C-library regional information | Установка / запрос C-библиотека региональная информация |
| setof/3 | Find all unique solutions to a goal | Найти все уникальные решения для цели |
| shell/0 | Execute interactive subshell | Выполнить интерактивный подоболочку |
| shell/1 | Execute OS command | Выполнить команду ОС |
| shell/2 | Execute OS command | Выполнить команду ОС |
| show\_profile/1 | Show results of the profiler | Показать результаты профайлере |
| size\_file/2 | Get size of a file in characters | Получить размер файла в символах |
| size\_nb\_set/2 | Determine size of не-backtrackable set | Определить размер не-backtrackable набора |
| skip/1 | Skip to character in current input | Перейти к характеру в текущем входе |
| skip/2 | Skip to character on stream | Перейти к характеру на поток |
| rl\_add\_history/1 | Add line to readline(3) history | Добавить строку в историю чтения строк |
| rl\_read\_history/1 | Read readline(3) history | Прочитать историю чтениия строк |
| rl\_read\_init\_file/1 | Read readline(3) init file | Прочитать файл инициализации чтения строк |
| rl\_write\_history/1 | Write readline(3) history | Записать историю чтения строк |
| sleep/1 | Suspend execution for specified time | Задержать выполнение в течение заданного времени |
| sort/2 | Sort elements in a list | Сортировка элементов в списке |
| sort/4 | Sort elements in a list | Сортировка элементов в списке |
| source\_exports/2 | Check whether source exports a predicate | Проверка экспортирует ли источник предикат |
| source\_file/1 | Examine currently loaded source files | Проверка загруженных исходных файлов |
| source\_file/2 | Obtain source file of predicate | Получить исходный файл предиката |
| source\_file\_property/2 | Information about loaded files | Информация о загруженных файлах |
| source\_location/2 | Location of last read term | Место последнего чтения утверждения |
| split\_string/4 | Break a string into substrings | Разбить строку на подстроки |
| spy/1 | Force tracer on specified predicate | Принудительная трассировка указанного предиката |
| stamp\_date\_time/3 | Convert time-stamp to date structure | Преобразование временной штамп на сегодняшний день структура |
| statistics/0 | Show execution statistics | Показать статистику выполнения |
| statistics/2 | Obtain collected statistics | Получить собранную статистику |
| stream\_pair/3 | Create/examine a bi-directional stream | Создание/проверить двунаправленный поток |
| stream\_position\_data/3 | Access fields from stream position | Поля доступа из позиции потока |
| stream\_property/2 | Get stream properties | Получить свойства потока |
| string/1 | Type check for string | Проверка типа для строки |
| string\_concat/3 | atom\_concat/3 for strings | Конкатанация (слияние) атомов для строк |
| string\_length/2 | Determine length of a string | Определить длину строки |
| string\_chars/2 | Conversion between string and list of characters | Преобразование между строкой и списком символов |
| string\_codes/2 | Conversion between string and list of character codes | Преобразование между строкой и списком кодов символов |
| string\_code/3 | Get or find a character code in a string | Получить или найти код символа в строке |
| string\_lower/2 | Case conversion to lower case | Преобразование в нижний регистр |
| string\_upper/2 | Case conversion to upper case | Преобразование в верхний регистр |
| string\_predicate/1 | (hook) Predicate contains strings | (хук) Предикат содержит строки |
| strip\_module/3 | Extract context module and term | Извлечь модуль контекста и утверждения |
| style\_check/1 | Change level of warnings | Изменение уровня предупреждений |
| sub\_atom/5 | Take a substring from an atom | Извлечь подстроку из атома |
| sub\_atom\_icasechk/3 | Case insensitive substring match | Учитиывать регистр подстроки |
| sub\_string/5 | Take a substring from a string | Извлечь подстроку из строки |
| subsumes\_term/2 | One-sided unification test | Односторонний тест объединение |
| succ/2 | Logical integer successor relation | Логическое целое преемник отношения |
| swritef/2 | Formatted write on a string | Форматированная запись в строку |
| swritef/3 | Formatted write on a string | Форматированная запись в строку |
| tab/1 | Output number of spaces | Выходное количество пробелов |
| tab/2 | Output number of spaces on a stream | Выходной количество пробелов в поток |
| tdebug/0 | Switch all threads into debug mode | Переключить все потоки в режиме отладки |
| tdebug/1 | Switch a thread into debug mode | Переключить поток в режим отладки |
| tell/1 | Change current output stream | Изменение текущей выходной поток |
| telling/1 | Query current output stream | Запрос текущего выходного потока |
| term\_expansion/2 | (hook) Convert term before compilation | (хук) Преобразовать утверждение перед компиляцией |
| term\_expansion/4 | (hook) Convert term before compilation | (хук) Преобразовать утверждение перед компиляцией |
| term\_string/2 | Read/write a term from/to a string | Прочитать / запись утверждение из / в строку |
| term\_string/3 | Read/write a term from/to a string | Прочитать / запись утверждение из / в строку |
| term\_subsumer/3 | Most specific generalization of two terms | Наиболее конкретные обобщение двух утверждений |
| term\_to\_atom/2 | Convert between term and atom | Преобразование между утверждением и атомом |
| thread\_at\_exit/1 | Register goal to be called at exit | Зарегистрировать цель вызываемое при выходе |
| thread\_create/3 | Create a new Prolog task | Создать новую задачу Пролог |
| thread\_detach/1 | Make thread cleanup after completion | Очистить поток после завершения |
| thread\_exit/1 | Terminate Prolog task with value | Завершить задачу Пролога со значением |
| thread\_get\_message/1 | Wait for message | Дождаться сообщения |
| thread\_get\_message/2 | Wait for message in a queue | Дождаться сообщения в очереди |
| thread\_get\_message/3 | Wait for message in a queue | Дождаться сообщения в очереди |
| thread\_initialization/1 | Run action at start of thread | Запустить действие при старте потока |
| thread\_join/2 | Wait for Prolog task-completion | Дождиться завершения задачи Пролога |
| thread\_local/1 | Declare thread-specific clauses for a predicate | Объявление поток-специфичного положения для предиката |
| thread\_message\_hook/3 | Thread local message\_hook/3 | Локальный хук сообщений утверждений / 3 |
| thread\_peek\_message/1 | Test for message | Тест на сообщение |
| thread\_peek\_message/2 | Test for message in a queue | Тест на сообщения в очереди |
| thread\_property/2 | Examine Prolog threads | Изучение потоков Пролога |
| thread\_self/1 | Get identifier of current thread | Получить идентификатор текущего потока |
| thread\_send\_message/2 | Send message to another thread | Отправить сообщение в другой поток |
| thread\_send\_message/3 | Send message to another thread | Отправить сообщение в другой поток |
| thread\_setconcurrency/2 | Number of active threads | Количество активных потоков |
| thread\_signal/2 | Execute goal in another thread | Выполнить цель в другом потоке |
| thread\_statistics/3 | Get statistics of another thread | Получить статистику другого потока |
| threads/0 | List running threads | Список запущенных потоков |
| throw/1 | Raise an exception (see catch/3) | Сгенерировать исключение (смотри оператор catch / 3) |
| time/1 | Determine time needed to execute goal | Определить время, необходимое для выполнения задачи |
| time\_file/2 | Get last modification time of file | Получить время последнего изменения файла |
| tmp\_file/2 | Create a temporary filename | Создание временного файла |
| tmp\_file\_stream/3 | Create a temporary file and open it | Создание временного файла и его открытие |
| tnodebug/0 | Switch off debug mode in all threads | Выключить режим отладки во всех потоках |
| tnodebug/1 | Switch off debug mode in a thread | Выключить режим отладки в потоке |
| told/0 | Close current output | Закрыть текущий вывод |
| tprofile/1 | Profile a thread for some period | Профиль потока в течение некоторого периода |
| trace/0 | Start the tracer | Запустить трейсер |
| trace/1 | Set trace point on predicate | Установить точку трассировки на предикат |
| trace/2 | Set/Clear trace point on ports | Установить/Очистить точку трассировки на портах |
| tracing/0 | Query status of the tracer | Запрос статуса трейсера |
| trim\_stacks/0 | Release unused memory resources | Освободить неиспользованные ресурсы памяти |
| true/0 | Succeed | Успешно |
| tspy/1 | Set spy point and enable debugging in all threads | Установите точку и включить отладку во всех потоках |
| tspy/2 | Set spy point and enable debugging in a thread | Установите точку и включить отладку в потоке |
| tty\_get\_capability/3 | Get terminal parameter | Получить параметр терминала |
| tty\_goto/2 | Goto position on screen | Установить точку перехода на экране |
| tty\_put/2 | Write control string to terminal | Записать строку управления на терминал |
| tty\_size/2 | Get row/column size of the terminal | Получить размер строки / столбца утвержденияла |
| ttyflush/0 | Flush output on terminal | Скрытый вывод на терминал |
| unify\_with\_occurs\_check/2 | Logically sound unification | Логически обоснованное объединение |
| unifiable/3 | Determining binding required for unification | Определение привязки, необходимой для унификации |
| unix/1 | OS interaction | взаимодействие с ОС |
| unknown/2 | Trap undefined predicates | Ловушка для неопределенных предикатов |
| unload\_file/1 | Unload a source file | Выгрузить исходный файл |
| unload\_foreign\_library/1 | library(shlib) Detach shared library (.so file) | библиотека (shlib) отсоединить разделяемую библиотеку (.so файл) |
| unload\_foreign\_library/2 | library(shlib) Detach shared library (.so file) | библиотека (shlib) отсоединить разделяемую библиотеку (.so файл) |
| unsetenv/1 | Delete shell environment variable | Удалить переменную окружения оболочки |
| upcase\_atom/2 | Convert atom to upper-case | Преобразовать атом в верхний регистр |
| use\_foreign\_library/1 | Load DLL/shared object (directive) | Загрузить DLL / общий объект (директива) |
| use\_foreign\_library/2 | Load DLL/shared object (directive) | Загрузить DLL / общий объект (директива) |
| use\_module/1 | Import a module | Импорт модуля |
| use\_module/2 | Import predicates from a module | Импорт предикатов из модуля |
| valid\_string\_goal/1 | (hook) Goal handles strings | (Хук) Цель обработки строк |
| var/1 | Type check for unbound variable | Проверка типа для несвязанных переменных |
| var\_number/2 | Check that var is numbered by numbervars | Проверка, что переменная нумеруется numbervars |
| var\_property/2 | Variable properties during macro expansion | Свойства переменных во время макроподстановок |
| variant\_sha1/2 | Term-hash for term-variants | хеш утверждений для вариантов утверждений |
| variant\_hash/2 | Term-hash for term-variants | хеш утверждений для вариантов утверждений |
| version/0 | Print system banner message | Системы печати заглавного сообщения |
| version/1 | Add messages to the system banner | Добавление сообщений в системное заглавное сообщение |
| visible/1 | Ports that are visible in the tracer | Порты, которые видны в трассере |
| volatile/1 | Predicates that are not saved | Предикаты, которые не сохраняются |
| wait\_for\_input/3 | Wait for input with optional timeout | Дождитесь ввода с дополнительным тайм-аут |
| when/2 | Execute goal when condition becomes true | Выполнить цель, когда условие становится истинным |
| wildcard\_match/2 | Csh(1) style wildcard match | Csh (1) Проверка соответствия запроса шаблону |
| win\_add\_dll\_directory/1 | Add directory to DLL search path | Добавить каталог DLL путь поиска |
| win\_add\_dll\_directory/2 | Add directory to DLL search path | Добавить каталог DLL путь поиска |
| win\_remove\_dll\_directory/1 | Remove directory from DLL search path | Удалить каталог из пути поиска DLL |
| win\_exec/2 | Win32: spawn Windows task | Win32: создать задачу Windows |
| win\_has\_menu/0 | Win32: true if console menu is available | Win32: верно если меню консоли доступно |
| win\_folder/2 | Win32: get special folder by CSIDL | Win32: получить специальную папку CSIDL |
| win\_insert\_menu/2 | swipl-win.exe: add menu | swipl-win.exe: добавить меню |
| win\_insert\_menu\_item/4 | swipl-win.exe: add item to menu | swipl-win.exe: добавить пункт в меню |
| win\_shell/2 | Win32: open document through Shell | Win32: открыть документ через оболочку |
| win\_shell/3 | Win32: open document through Shell | Win32: открыть документ через оболочку |
| win\_registry\_get\_value/3 | Win32: get registry value | Win32: получить значение реестра |
| win\_window\_pos/1 | Win32: change size and position of window | Win32: изменить размер и положение окна |
| window\_title/2 | Win32: change title of window | Win32: изменить заголовок окна |
| with\_mutex/2 | Run goal while holding mutex | Запустить цель, удерживая мьютекс |
| with\_output\_to/2 | Write to strings and more | Записать в строки и многое другое |
| with\_quasi\_quotation\_input/3 | Parse quasi quotation from stream | Разбор квази цитаты из потока |
| working\_directory/2 | Query/change CWD | Запросить/изменить CWD |
| write/1 | Write term | Записать утверждение |
| write/2 | Write term to stream | Записать утверждение для потоковой передачи |
| writeln/1 | Write term, followed by a newline | Записать утверждение с последующим переводом строки |
| writeln/2 | Write term, followed by a newline to a stream | Записать утверждение за которым следует символ новой строки в поток |
| write\_canonical/1 | Write a term with quotes, ignore operators | Записать утверждение в кавычки, игнорируя операторы |
| write\_canonical/2 | Write a term with quotes, ignore operators on a stream | Записать утверждение в кавычки, игнорируя операторы в поток |
| write\_length/3 | Dermine #characters to output a term | Определить символы для вывода утверждения |
| write\_term/2 | Write term with options | Записать утверждение с параметрами |
| write\_term/3 | Write term with options to stream | Записать утверждение с параметрами для потоковой передачи |
| writef/1 | Formatted write | Форматированный вывод |
| writef/2 | Formatted write on stream | Форматированный вывод в поток |
| writeq/1 | Write term, insert quotes | Записать утверждение, вставив кавычки |
| writeq/2 | Write term, insert quotes on stream | Записать утверждение, вставив кавычки в поток |

Приложение А

(обязательное)

Примеры программ

Пример 1. Hello, World!:

[Пример](http://progopedia.ru/example/hello-world/97/) для версий [B-Prolog 7.4-3](http://progopedia.ru/version/b-prolog-7.4-3/), [ECLiPSe CLP 6.0 #188](http://progopedia.ru/version/eclipse-clp-6.0-188/), [Poplog 15.5 (Prolog)](http://progopedia.ru/version/poplog-15.5-prolog/), [gprolog 1.3.0](http://progopedia.ru/version/gprolog-1.3.0/), [swipl 5.6.x](http://progopedia.ru/version/swipl-5.6.x/)

write('Hello, World!'), nl.

Этот пример не требует загрузки фактов или правил. Запрос выполняется в интерактивном режиме, и его результат выглядит следующим образом:

Hello, World!

yes

Первая строка является собственно выводом предиката write, вторая — результат оценивания запроса.

Следует отметить, что замена одинарных кавычек на двойные выводит строку как массив ASCII-кодов отдельных символов:

| ?- write("Hello, World!").

[72,101,108,108,111,44,32,87,111,114,108,100,33]

yes

Пример 2.Квадратное уравнение:

[Пример](http://progopedia.ru/example/quadratic-equation/218/) для версий [B-Prolog 7.4-3](http://progopedia.ru/version/b-prolog-7.4-3/), [gprolog 1.3.0](http://progopedia.ru/version/gprolog-1.3.0/), [swipl 5.6.x](http://progopedia.ru/version/swipl-5.6.x/)

Этот пример соответствует стандарту [ISO Prolog](http://progopedia.ru/dialect/iso-prolog/) и использует встроенный предикат read/1. Следует отметить, что при вводе термов этим способом после каждого терма следует ставить точку.

ISO Prolog — стандартизированный диалект языка программирования Prolog.

Стандарт ISO Prolog состоит из двух частей. ISO/IEC 13211-1 был опубликован в 1995 году и ставил своей целью стандартизацию существующих реализаций основных элементов языка. Этот документ прояснил аспекты языка, которые до того были неоднозначными и препятствовали портированию программ на неродные реализации.

ISO/IEC 13211-2, опубликованный в 2000 году, добавляет к стандарту поддержку модулей.

[Материалы по стандарту ISO Prolog](http://pauillac.inria.fr/~deransar/prolog/docs.html): http://www.deransart.fr//prolog/docs.html

q :- write('A = '),

read(A),

( A = 0, write('Not a quadratic equation');

write('B = '),

read(B),

write('C = '),

read(C),

D is B\*B-4\*A\*C,

(D = 0, write('x = '), X is -B/2/A, write(X);

D > 0, write('x1 = '), X1 is (-B+sqrt(D))/2/A, write(X1), nl, write('x2 = '), X2 is (-B-sqrt(D))/2/A, write(X2);

R is -B/2/A, I is abs(sqrt(-D)/2/A),

write('x1 = ('), write(R), write(', '), write(I), write(')'), nl,

write('x1 = ('), write(R), write(', -'), write(I), write(')')

)

).

Пример 3.Факториал:

[Пример](http://progopedia.ru/example/factorial/215/) для версий [B-Prolog 7.4-3](http://progopedia.ru/version/b-prolog-7.4-3/), [gprolog 1.3.0](http://progopedia.ru/version/gprolog-1.3.0/), [swipl 5.6.x](http://progopedia.ru/version/swipl-5.6.x/)

Как в [GNU Prolog](http://progopedia.ru/implementation/gnu-prolog/), так и в [B-Prolog](http://progopedia.ru/implementation/b-prolog/) 12! не помещается в целочисленный тип данных, поэтому все значения после 11! неправильны. В [SWI-Prolog](http://progopedia.ru/implementation/swi-prolog/) переполнения не возникает.

`| ?- fact(16,X).

0! = 1

1! = 1

2! = 2

3! = 6

4! = 24

5! = 120

6! = 720

7! = 5040

8! = 40320

9! = 362880

10! = 3628800

11! = 39916800

12! = -57869312

13! = -215430144

14! = 205203456

15! = -143173632

16! = -143294464

X = -143294464 ?`

% fact.pl

fact(X, F) :-

( X=0, F=1;

Y is X-1, fact(Y, Z), F is X\*Z),

write(X), write('! = '), write(F), nl.

% interactive

[fact].

fact(16,X).

Пример 4.Числа Фибоначчи:

[Пример](http://progopedia.ru/example/fibonacci/216/) для версий [B-Prolog 7.4-3](http://progopedia.ru/version/b-prolog-7.4-3/), [gprolog 1.3.0](http://progopedia.ru/version/gprolog-1.3.0/), [swiprolog 5.6.x](http://progopedia.ru/version/swipl-5.6.x/)

Пример почти идентичен примеру для [Poplog](http://progopedia.ru/example/fibonacci/98/) , за исключением синтаксиса подключения файла.

% fibonacci.pl

:- dynamic(stored/1).

memo(Goal) :-

stored(Goal) -> true;

Goal, assertz(stored(Goal)).

fib(1,1) :- !, write('1, ').

fib(2,1) :- !, write('1, ').

fib(N,F) :-

N1 is N-1, memo(fib(N1,F1)),

N2 is N-2, memo(fib(N2,F2)),

F is F1 + F2,

write(F), write(', ').

% interactive

[fibonacci].

fib(16,X), write('...'), nl.

Приложение В

(обязательное)

Описание основных предикатов SWI-пролог

**abolish**(*:PredicateIndicator*)

Removes all clauses of a predicate with functor *Functor* and arity *Arity* from the database. All predicate attributes (dynamic, multifile, index, etc.) are reset to their defaults. Abolishing an imported predicate only removes the import link; the predicate will keep its old definition in its definition module.

According to the ISO standard, [abolish/1](http://www.swi-prolog.org/pldoc/man?predicate=abolish/1) can only be applied to dynamic procedures. This is odd, as for dealing with dynamic procedures there is already [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1) and [retractall/1](http://www.swi-prolog.org/pldoc/man?predicate=retractall/1). The[abolish/1](http://www.swi-prolog.org/pldoc/man?predicate=abolish/1) predicate was introduced in DEC-10 Prolog precisely for dealing with static procedures. In SWI-Prolog, [abolish/1](http://www.swi-prolog.org/pldoc/man?predicate=abolish/1) works on static procedures, unless the Prolog flag [iso](http://www.swi-prolog.org/pldoc/man?section=flags#flag:iso) is set to true.

It is advised to use [retractall/1](http://www.swi-prolog.org/pldoc/man?predicate=retractall/1) for erasing all clauses of a dynamic predicate.

**abolish**(*+Name, +Arity*)

Same as abolish(Name/Arity). The predicate [abolish/2](http://www.swi-prolog.org/pldoc/man?predicate=abolish/2) conforms to the Edinburgh standard, while [abolish/1](http://www.swi-prolog.org/pldoc/man?predicate=abolish/1) is ISO compliant.

**abort**

Abort the Prolog execution and restart the top level. If the **-t** *toplevel* command line option is given, this goal is started instead of entering the default interactive top level.

Aborting is implemented by throwing the reserved exception '$aborted'. This exception can be caught using [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3), but the recovery goal is wrapped with a predicate that prunes the choice points of the recovery goal (i.e., as [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1)) and re-throws the exception. This is illustrated in the example below, where we press control-C and `a'.

?- catch((repeat,fail), E, true).

^CAction (h for help) ? abort

% Execution Aborted

**absolute\_file\_name**(*+File, -Absolute*)

Expand a local filename into an absolute path. The absolute path is canonicalised: references to . and .. are deleted. This predicate ensures that expanding a filename returns the same absolute path regardless of how the file is addressed. SWI-Prolog uses absolute filenames to register source files independent of the current working directory. See also[absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3). See also [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3) and [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2).

**absolute\_file\_name**(*+Spec, -Absolute, +Options*)

Convert the given file specification into an absolute path. *Spec* is a term Alias(Relative) (e.g.,(library(lists)), a relative filename or an absolute filename. The primary intention of this predicate is to resolve files specified as Alias(Relative). *Option* is a list of options to guide the conversion:

**extensions**(*ListOfExtensions*)

List of file extensions to try. Default is ''. For each extension, [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3) will first add the extension and then verify the conditions imposed by the other options. If the condition fails, the next extension on the list is tried. Extensions may be specified both as .ext or plain ext.

**relative\_to**(*+FileOrDir*)

Resolve the path relative to the given directory or the directory holding the given file. Without this option, paths are resolved relative to the working directory (see[working\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=working_directory/2)) or, if *Spec* is atomic and absolute\_file\_name/[2,3] is executed in a directive, it uses the current source file as reference.

**access**(*Mode*)

Imposes the condition access\_file(*File*, *Mode*). *Mode* is one of read, write, append,execute, exist or none. See also [access\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=access_file/2).

**file\_type**(*Type*)

Defines extensions. Current mapping: txt implies [''], prolog implies ['.pl', ''],executable implies ['.so', ''], qlf implies ['.qlf', ''] and directory implies['']. The file type source is an alias for prolog for compatibility with SICStus Prolog. See also [prolog\_file\_type/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_file_type/2). This predicate only returns non-directories, unless the option file\_type(directory) is specified.

**file\_errors**(*fail/error*)

If error (default), throw an existence\_error exception if the file cannot be found. Iffail, stay silent.122

**solutions**(*first/all*)

If first (default), the predicate leaves no choice point. Otherwise a choice point will be left and backtracking may yield more solutions.

**expand**(*true/false*)

If true (default is false) and *Spec* is atomic, call [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2) followed by[member/2](http://www.swi-prolog.org/pldoc/man?predicate=member/2) on *Spec* before proceeding. This is a SWI-Prolog extension.

The Prolog flag [verbose\_file\_search](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose_file_search) can be set to true to help debugging Prolog's search for files.

This predicate is derived from Quintus Prolog. In Quintus Prolog, the argument order wasabsolute\_file\_name(+Spec, +Options, -Path). The argument order has been changed for compatibility with ISO and SICStus. The Quintus argument order is still accepted.

**access\_file**(*+File, +Mode*)

True if *File* exists and can be accessed by this Prolog process under mode *Mode*. *Mode* is one of the atoms read, write, append, exist, none or execute. *File* may also be the name of a directory. Fails silently otherwise. access\_file(File, none) simply succeeds without testing anything.

If *Mode* is write or append, this predicate also succeeds if the file does not exist and the user has write access to the directory of the specified location.

The bahaviour is backed up by the POSIX access() API. The Windows replacement (\_waccess()) returns incorrect results because it does not consider ACLs (Access Control Lists). The Prolog flag [win\_file\_access\_check](http://www.swi-prolog.org/pldoc/man?section=flags#flag:win_file_access_check) may be used to control the level of checking performed by Prolog. Please note that checking access never provides a guarantee that a subsequent open succeeds without errors due to inherent concurrency in file operations. It is generally more robust to try and open the file and handle possible exceptions. See [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) and [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3).

**acyclic\_term**(*@Term*)

True if *Term* does not contain cycles, i.e. can be processed recursively in finite time. See also[cyclic\_term/1](http://www.swi-prolog.org/pldoc/man?predicate=cyclic_term/1) and [section 2.16](http://www.swi-prolog.org/pldoc/man?section=cyclic).

**add\_import\_module**(*+Module, +Import, +StartOrEnd*)

If *Import* is not already an import module for *Module*, add it to this list at the start or enddepending on *StartOrEnd*. See also [import\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=import_module/2) and [delete\_import\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=delete_import_module/2).

**add\_nb\_set**(*+Key, !Set*)

Add *Key* to *Set*. If *Key* is already a member of *Set*, [add\_nb\_set/3](http://www.swi-prolog.org/pldoc/man?predicate=add_nb_set/3) succeeds without modifying*Set*.

**add\_nb\_set**(*+Key, !Set, ?New*)

If *Key* is not in *Set* and *New* is unified to true, *Key* is added to *Set*. If *Key* is in *Set*, *New* is unified to false. It can be used for many purposes:

|  |  |
| --- | --- |
| add\_nb\_set(+, +, false) | Test membership |
| add\_nb\_set(+, +, true) | Succeed only if new member |
| add\_nb\_set(+, +, Var) | Succeed, binding *Var* |

**append**(*+File*)

Similar to [tell/1](http://www.swi-prolog.org/pldoc/man?predicate=tell/1), but positions the file pointer at the end of *File* rather than truncating an existing file. The pipe construct is not accepted by this predicate.

**apply**(*:Goal, +List*)

Append the members of *List* to the arguments of *Goal* and call the resulting term. For example: apply(plus(1), [2, X]) calls plus(1, 2, X). New code should use call/[2..] if the length of *List* is fixed.

**apropos**(*+Pattern*)

Display all predicates, functions and sections that have *Pattern* in their name or summary description. Lowercase letters in *Pattern* also match a corresponding uppercase letter. Example:

|  |  |
| --- | --- |
| ?- apropos(file). | Display predicates, functions and sections that have `file' (or `File', etc.) in their summary description. |

**arg**(*?Arg, +Term, ?Value*)

*Term* should be instantiated to a term, *Arg* to an integer between 1 and the arity of *Term*. *Value*is unified with the *Arg*-th argument of *Term*. *Arg* may also be unbound. In this case *Value* will be unified with the successive arguments of the term. On successful unification, *Arg* is unified with the argument number. Backtracking yields alternative solutions.86 The predicate [arg/3](http://www.swi-prolog.org/pldoc/man?predicate=arg/3)fails silently if *Arg = 0* or *Arg > arity* and raises the exceptiondomain\_error(not\_less\_than\_zero, *Arg*) if *Arg < 0*.

**assoc\_to\_list**(*+Assoc, -List*)

*List* is a list of Key-Value pairs corresponding to the associations in *Assoc* in ascending order of keys.

**assert**(*+Term*)

Equivalent to [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1). Deprecated: new code should use [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1).

**assert**(*+Term, -Reference*)

Equivalent to [assertz/2](http://www.swi-prolog.org/pldoc/man?predicate=assertz/2). Deprecated: new code should use [assertz/2](http://www.swi-prolog.org/pldoc/man?predicate=assertz/2).

**asserta**(*+Term*)

Assert a fact or clause in the database. *Term* is asserted as the first fact or clause of the corresponding predicate. Equivalent to [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1), but *Term* is asserted as first clause or fact of the predicate. If the program space for the target module is limited (see [set\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=set_module/1)),[asserta/1](http://www.swi-prolog.org/pldoc/man?predicate=asserta/1) can raise a resource\_error(program\_space).

**asserta**(*+Term, -Reference*)

Asserts a clause as [asserta/1](http://www.swi-prolog.org/pldoc/man?predicate=asserta/1) and unifies *Reference* with a handle to this clause. The handle can be used to access this specific clause using [clause/3](http://www.swi-prolog.org/pldoc/man?predicate=clause/3) and [erase/1](http://www.swi-prolog.org/pldoc/man?predicate=erase/1).

**assertion**(*:Goal*)

Acts similar to C assert() macro. It has no effect if *Goal* succeeds. If *Goal* fails or throws an exception, the following steps are taken:

* call [**prolog:assertion\_failed/2**](http://www.swi-prolog.org/pldoc/debug.html#prolog:assertion_failed/2). If [**prolog:assertion\_failed/2**](http://www.swi-prolog.org/pldoc/debug.html#prolog:assertion_failed/2) fails, then:
  + If this is an interactive toplevel thread, print a message, the stack-trace, and finally trap the debugger.
  + Otherwise, throw error(assertion\_error(Reason, G),\_) where Reason is one of fail or the exception raised.

**assertz**(*+Term*)

Equivalent to [asserta/1](http://www.swi-prolog.org/pldoc/man?predicate=asserta/1), but *Term* is asserted as the last clause or fact of the predicate.

**assertz**(*+Term, -Reference*)

Equivalent to [asserta/1](http://www.swi-prolog.org/pldoc/man?predicate=asserta/1), asserting the new clause as the last clause of the predicate.

**attach\_console**

If the current thread has no console attached yet, attach one and redirect the user streams (input, output, and error) to the new console window. On Unix systems the console is an**xterm** application. On Windows systems this requires the GUI version **swipl-win.exe** rather than the console-based **swipl.exe**.

This predicate has a couple of useful applications. One is to separate (debugging) I/O of different threads. Another is to start debugging a thread that is running in the background. If thread 10 is running, the following sequence starts the tracer on this thread:

?- thread\_signal(10, (attach\_console, trace)).

**attribute\_goals**(*+Var*)//

This nonterminal, if it is defined in a module, is used by [copy\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/3) to project attributes of that module to residual goals. It is also used by the top level to obtain residual goals after executing a query.

**attr\_unify\_hook**(*+AttValue, +VarValue*)

A hook that must be defined in the module to which an attributed variable refers. It is called*after* the attributed variable has been unified with a non-var term, possibly another attributed variable. *AttValue* is the attribute that was associated to the variable in this module and*VarValue* is the new value of the variable. If this predicate fails, the unification fails. If *VarValue*is another attributed variable the hook often combines the two attributes and associates the combined attribute with *VarValue* using [put\_attr/3](http://www.swi-prolog.org/pldoc/man?predicate=put_attr/3).

**attr\_portray\_hook**(*+AttValue, +Var*)

Called by [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) and friends for each attribute if the option attributes(portray) is in effect. If the hook succeeds the attribute is considered printed. Otherwise Module = ... is printed to indicate the existence of a variable. New infrastructure dealing with communicating attribute values must be based on [copy\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/3) and its hook [attribute\_goals/3](http://www.swi-prolog.org/pldoc/man?predicate=attribute_goals/3).

**attvar**(*@Term*)

Succeeds if *Term* is an attributed variable. Note that [var/1](http://www.swi-prolog.org/pldoc/man?predicate=var/1) also succeeds on attributed variables. Attributed variables are created with [put\_attr/3](http://www.swi-prolog.org/pldoc/man?predicate=put_attr/3).

**at\_end\_of\_stream**

Succeeds after the last character of the current input stream has been read. Also succeeds if there is no valid current input stream.

**at\_end\_of\_stream**(*+Stream*)

Succeeds after the last character of the named stream is read, or *Stream* is not a valid input stream. The end-of-stream test is only available on buffered input streams (unbuffered input streams are rarely used; see [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4)).

**at\_halt**(*:Goal*)

Register *Goal* to be run from [PL\_cleanup()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_cleanup), which is called when the system halts. The hooks are run in the reverse order they were registered (FIFO). Success or failure executing a hook is ignored. If the hook raises an exception this is printed using [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). An attempt to call halt/[0,1] from a hook is ignored. Hooks may call [cancel\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=cancel_halt/1), causing [halt/0](http://www.swi-prolog.org/pldoc/man?predicate=halt/0) and[PL\_halt(0)](http://www.swi-prolog.org/pldoc/man?CAPI=PL_halt) to print a message indicating that halting the system has been cancelled.

**atom**(*@Term*)

True if *Term* is bound to an atom.

**atom\_chars**(*?Atom, ?CharList*)

As [atom\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_codes/2), but *CharList* is a list of one-character atoms rather than a list of character codes.89

?- atom\_chars(hello, X).

X = [h, e, l, l, o]

**atom\_codes**(*?Atom, ?String*)

Convert between an atom and a list of character codes. If *Atom* is instantiated, it will be translated into a list of character codes and the result is unified with *String*. If *Atom* is unbound and *String* is a list of character codes, *Atom* will be unified with an atom constructed from this list.

**atom\_concat**(*?Atom1, ?Atom2, ?Atom3*)

*Atom3* forms the concatenation of *Atom1* and *Atom2*. At least two of the arguments must be instantiated to atoms. This predicate also allows for the mode (-,-,+), non-deterministically splitting the 3rd argument into two parts (as [append/3](http://www.swi-prolog.org/pldoc/man?predicate=append/3) does for lists). SWI-Prolog allows for atomic arguments. Portable code must use [atomic\_concat/3](http://www.swi-prolog.org/pldoc/man?predicate=atomic_concat/3) if non-atom arguments are involved.

**atom\_length**(*+Atom, -Length*)

True if *Atom* is an atom of *Length* characters. The SWI-Prolog version accepts all atomic types, as well as code-lists and character-lists. New code should avoid this feature and use[write\_length/3](http://www.swi-prolog.org/pldoc/man?predicate=write_length/3) to get the number of characters that would be written if the argument was handed to [write\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=write_term/3).

**atom\_number**(*?Atom, ?Number*)

Realises the popular combination of [atom\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_codes/2) and [number\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=number_codes/2) to convert between atom and number (integer or float) in one predicate, avoiding the intermediate list. Unlike the ISO [number\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=number_codes/2) predicates, [atom\_number/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_number/2) fails silently in mode (+,-) if *Atom* does not represent a number.92 See also [atomic\_list\_concat/2](http://www.swi-prolog.org/pldoc/man?predicate=atomic_list_concat/2) for assembling an atom from atoms and numbers.

**atom\_prefix**(*+Atom, +Prefix*)

True if *Atom* starts with the characters from *Prefix*. Its behaviour is equivalent to ?- sub\_atom(*Atom*, 0, \_, \_, *Prefix*). Deprecated.

**atom\_string**(*?Atom, ?String*)

Bi-directional conversion between an atom and a string. At least one of the two arguments must be instantiated. *Atom* can also be an integer or floating point number.

**atom\_to\_term**(*+Atom, -Term, -Bindings*)

Use *Atom* as input to [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) using the option variable\_names and return the read term in *Term* and the variable bindings in *Bindings*. *Bindings* is a list of *Name = Var* couples, thus providing access to the actual variable names. See also [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2). If *Atom* has no valid syntax, a syntax\_error exception is raised. New code should use [read\_term\_from\_atom/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term_from_atom/3).

**atomic**(*@Term*)

True if *Term* is bound (i.e., not a variable) and is not compound. Thus, atomic acts as if defined by:

atomic(Term) :-

nonvar(Term),

\+ compound(Term).

SWI-Prolog defines the following atomic datatypes: atom ([atom/1](http://www.swi-prolog.org/pldoc/man?predicate=atom/1)), string ([string/1](http://www.swi-prolog.org/pldoc/man?predicate=string/1)), integer ([integer/1](http://www.swi-prolog.org/pldoc/man?predicate=integer/1)), floating point number ([float/1](http://www.swi-prolog.org/pldoc/man?predicate=float/1)) and blob ([blob/2](http://www.swi-prolog.org/pldoc/man?predicate=blob/2)). In addition, the symbol [](empty list) is atomic, but not an atom. See [section 5.1](http://www.swi-prolog.org/pldoc/man?section=ext-lists).

**atomic\_concat**(*+Atomic1, +Atomic2, -Atom*)

*Atom* represents the text after converting *Atomic1* and *Atomic2* to text and concatenating the result:

?- atomic\_concat(name, 42, X).

X = name42.

**atomic\_list\_concat**(*+List, -Atom*)

*List* is a list of strings, atoms, integers or floating point numbers. Succeeds if *Atom* can be unified with the concatenated elements of *List*. Equivalent to atomic\_list\_concat(List, '', Atom).

**atomic\_list\_concat**(*+List, +Separator, -Atom*)

Creates an atom just like [atomic\_list\_concat/2](http://www.swi-prolog.org/pldoc/man?predicate=atomic_list_concat/2), but inserts *Separator* between each pair of inputs. For example:

?- atomic\_list\_concat([gnu, gnat], ', ', A).

A = 'gnu, gnat'

The SWI-Prolog version of this predicate can also be used to split atoms by instantiating*Separator* and *Atom* as shown below. We kept this functionality to simplify porting old SWI-Prolog code where this predicate was called [concat\_atom/3](http://www.swi-prolog.org/pldoc/man?predicate=concat_atom/3). When used in mode (-,+,+),*Separator* must be a non-empty atom. See also [split\_string/4](http://www.swi-prolog.org/pldoc/man?predicate=split_string/4).

?- atomic\_list\_concat(L, -, 'gnu-gnat').

L = [gnu, gnat]

**atomics\_to\_string**(*+List, -String*)

*List* is a list of strings, atoms, integers or floating point numbers. Succeeds if *String* can be unified with the concatenated elements of *List*. Equivalent to atomics\_to\_string(List, '', String).

**atomics\_to\_string**(*+List, +Separator, -String*)

Creates a string just like [atomics\_to\_string/2](http://www.swi-prolog.org/pldoc/man?predicate=atomics_to_string/2), but inserts *Separator* between each pair of inputs. For example:

?- atomics\_to\_string([gnu, "gnat", 1], ', ', A).

A = "gnu, gnat, 1"

**autoload**

Check the current Prolog program for predicates that are referred to, are undefined and have a definition in the Prolog library. Load the appropriate libraries.

This predicate is used by qsave\_program/[1,2] to ensure the saved state does not depend on availability of the libraries. The predicate [autoload/0](http://www.swi-prolog.org/pldoc/man?predicate=autoload/0) examines all clauses of the loaded program (obtained with [clause/2](http://www.swi-prolog.org/pldoc/man?predicate=clause/2)) and analyzes the body for referenced goals. Such an analysis cannot be complete in Prolog, which allows for the creation of arbitrary terms at runtime and the use of them as a goal. The current analysis is limited to the following:

* Direct goals appearing in the body
* Arguments of declared meta-predicates that are marked with an integer (0..9). See[meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1).

The analysis of meta-predicate arguments is limited to cases where the argument appears literally in the clause or is assigned using =/2 before the meta-call. That is, the following fragment is processed correctly:

...,

Goal = prove(Theory),

forall(current\_theory(Theory),

Goal)),

But, the calls to prove\_simple/1 and prove\_complex/1 in the example below are *not*discovered by the analysis and therefore the modules that define these predicates must be loaded explicitly using [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1),2.

...,

member(Goal, [ prove\_simple(Theory),

prove\_complex(Theory)

]),

forall(current\_theory(Theory),

Goal)),

It is good practice to use [gxref/0](http://www.swi-prolog.org/pldoc/man?predicate=gxref/0) to make sure that the program has sufficient declarations such that the analaysis tools can verify that all required predicates can be resolved and that all code is called. See [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1), [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1), [public/1](http://www.swi-prolog.org/pldoc/man?predicate=public/1) and prolog:called\_by/2.

**autoload\_path**(*+DirAlias*)

Add *DirAlias* to the libraries that are used by the autoloader. This extends the search pathautoload and reloads the library index. For example:

:- autoload\_path(library(http)).

If this call appears as a directive, it is term-expanded into a clause for user:file\_search\_path/2 and a directive calling [reload\_library\_index/0](http://www.swi-prolog.org/pldoc/man?predicate=reload_library_index/0). This keeps source information and allows for removing this directive.

**b\_getval**(*+Name, -Value*)

Get the value associated with the global variable *Name* and unify it with *Value*. Note that this unification may further instantiate the value of the global variable. If this is undesirable the normal precautions (double negation or [copy\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/2)) must be taken. The [b\_getval/2](http://www.swi-prolog.org/pldoc/man?predicate=b_getval/2)predicate generates errors if *Name* is not an atom or the requested variable does not exist.

**b\_set\_dict**(*+Key, !Dict, +Value*)

Destructively update the value associated with *Key* in *Dict* to *Value*. The update is trailed and undone on backtracking. This predicate raises an existence error if *Key* does not appear in*Dict*. The update semantics are equivalent to [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) and [b\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=b_setval/2).

**b\_setval**(*+Name, +Value*)

Associate the term *Value* with the atom *Name* or replace the currently associated value with*Value*. If *Name* does not refer to an existing global variable, a variable with initial value [] is created (the empty list). On backtracking the assignment is reversed.

**bagof**(*+Template, :Goal, -Bag*)

Unify *Bag* with the alternatives of *Template*. If *Goal* has free variables besides the one sharing with *Template*, [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3) will backtrack over the alternatives of these free variables, unifying*Bag* with the corresponding alternatives of *Template*. The construct +*Var*^*Goal* tells [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3)not to bind *Var* in *Goal*. [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3) fails if *Goal* has no solutions.

The example below illustrates [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3) and the **^** operator. The variable bindings are printed together on one line to save paper.

2 ?- listing(foo).

foo(a, b, c).

foo(a, b, d).

foo(b, c, e).

foo(b, c, f).

foo(c, c, g).

true.

3 ?- bagof(C, foo(A, B, C), Cs).

A = a, B = b, C = G308, Cs = [c, d] ;

A = b, B = c, C = G308, Cs = [e, f] ;

A = c, B = c, C = G308, Cs = [g].

4 ?- bagof(C, A^foo(A, B, C), Cs).

A = G324, B = b, C = G326, Cs = [c, d] ;

A = G324, B = c, C = G326, Cs = [e, f, g].

5 ?-

**between**(*+Low, +High, ?Value*)

*Low* and *High* are integers, *High >=Low*. If *Value* is an integer, *Low =<Value =<High*. When *Value*is a variable it is successively bound to all integers between *Low* and *High*. If *High* is inf orinfinite96 [between/3](http://www.swi-prolog.org/pldoc/man?predicate=between/3) is true iff *Value >=Low*, a feature that is particularly interesting for generating integers from a certain value.

**blob**(*@Term, ?Type*)

True if *Term* is a *blob* of type *Type*. See [section 10.4.7](http://www.swi-prolog.org/pldoc/man?section=foreigninclude).

**break**

Recursively start a new Prolog top level. This Prolog top level shares everything from the environment it was started in. Debugging is switched off on entering a break and restored on leaving one. The break environment is terminated by typing the system's end-of-file character (control-D). If that is somehow not functional, the term end\_of\_file. can be entered to return from the break environment. If the **-t** *toplevel* command line option is given, this goal is started instead of entering the default interactive top level ([prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0)).

Notably the gui based versions (**swipl-win** on Windows and MacOS) provide the menu**Run/New thread** that opens a new toplevel that runs concurrently with the initial toplevel. The concurrent toplevel can be used to examine the program, in particular global dynamic predicates. It can not access *global variables* or thread-local dynamic predicates (see[thread\_local/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_local/1)) of the main thread.

|  |  |
| --- | --- |
| **break\_hook/6** | *(hook)* Debugger hook |
|  |  |

**byte\_count**(*+Stream, -Count*)

Byte position in *Stream*. For binary streams this is the same as [character\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=character_count/2). For text files the number may be different due to multi-byte encodings or additional record separators (such as Control-M in Windows).

**call**(*:Goal*)

Invoke *Goal* as a goal. Note that clauses may have variables as subclauses, which is identical to[call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1).

|  |  |
| --- | --- |
| **call/[2..]** | Call with additional arguments |

**call\_cleanup**(*:Goal, +Catcher, :Cleanup*)

Same as setup\_call\_catcher\_cleanup(true, Goal, Catcher, Cleanup). The same warning as for [call\_cleanup/2](http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2) applies.

**call\_cleanup**(*:Goal, :Cleanup*)

Same as setup\_call\_cleanup(true, Goal, Cleanup). This is provided for compatibility with a number of other Prolog implementations only. Do not use [call\_cleanup/2](http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2) if you perform side-effects prior to calling that will be undone by *Cleanup*. Instead, use[setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3) with an appropriate first argument to perform those side-effects.

**call\_dcg**(*:DCGBody, ?State0, ?State*)

As [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3), but without type checking *State0* and *State*. This allows for using DCG rules for threading an arbitrary state variable. This predicate was introduced after type checking was added to [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3).62

A portable solution for threading state through a DCG can be implemented by wrapping the state in a list and use the DCG push-back facility. Subsequently, the following predicates may be used to access and modify the state:63

state(S), [S] --> [S].

state(S0, S), [S] --> [S0].

**call\_residue\_vars**(*:Goal, -Vars*)

Find residual attributed variables left by *Goal*. This predicate is intended for debugging programs using coroutining or constraints. Consider a program that poses contradicting constraints on a variable. Such programs should fail, but sometimes succeed because the constraint solver is too weak to detect the contradiction. Ideally, delayed goals and constraints are all executed at the end of the computation. The meta predicate[call\_residue\_vars/2](http://www.swi-prolog.org/pldoc/man?predicate=call_residue_vars/2) finds variables that are given attribute variables or whose attributes are modified by *Goal*, regardless of whether or not these variables are reachable from the arguments of *Goal*.141.

**call\_shared\_object\_function**(*+Handle, +Function*)

Call the named function in the loaded shared library. The function is called without arguments and the return value is ignored. Normally this function installs foreign language predicates using calls to [PL\_register\_foreign()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_register_foreign).

**call\_with\_depth\_limit**(*:Goal, +Limit, -Result*)

If *Goal* can be proven without recursion deeper than *Limit* levels, [call\_with\_depth\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_depth_limit/3)succeeds, binding *Result* to the deepest recursion level used during the proof. Otherwise,*Result* is unified with depth\_limit\_exceeded if the limit was exceeded during the proof, or the entire predicate fails if *Goal* fails without exceeding *Limit*.

The depth limit is guarded by the internal machinery. This may differ from the depth computed based on a theoretical model. For example, [true/0](http://www.swi-prolog.org/pldoc/man?predicate=true/0) is translated into an inline virtual machine instruction. Also, [repeat/0](http://www.swi-prolog.org/pldoc/man?predicate=repeat/0) is not implemented as below, but as a non-deterministic foreign predicate.

repeat.

repeat :-

repeat.

As a result, [call\_with\_depth\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_depth_limit/3) may still loop infinitely on programs that should theoretically finish in finite time. This problem can be cured by using Prolog equivalents to such built-in predicates.

This predicate may be used for theorem provers to realise techniques like *iterative deepening*. See also [call\_with\_inference\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_inference_limit/3). It was implemented after discussion with Steve Moyle[smoyle@ermine.ox.ac.uk](mailto:smoyle@ermine.ox.ac.uk).

**call\_with\_inference\_limit**(*:Goal, +Limit, -Result*)

Equivalent to call(Goal), but limits the number of inferences *for each solution of Goal*.58. Execution may terminate as follows:

* If *Goal* does *not* terminate before the inference limit is exceeded, *Goal* is aborted by injecting the exception inference\_limit\_exceeded into its execution. After termination of *Goal*, *Result* is unified with the atom inference\_limit\_exceeded. *Otherwise*,
* If *Goal* fails, [call\_with\_inference\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_inference_limit/3) fails.
* If *Goal* succeeds *without a choice point*, *Result* is unified with !.
* If *Goal* succeeds *with a choice point*, *Result* is unified with true.
* If *Goal* throws an exception, [call\_with\_inference\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_inference_limit/3) re-throws the exception.

An inference is defined as a call or redo on a predicate. Please note that some primitive built-in predicates are compiled to virtual machine instructions for which inferences are not counted. The execution of predicates defined in other languages (e.g., C, C++) count as a single inference. This includes potentially expensive built-in predicates such as [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2).

Calls to this predicate may be nested. An inner call that sets the limit below the current is honoured. An inner call that would terminate after the current limit does not change the effective limit. See also [call\_with\_depth\_limit/3](http://www.swi-prolog.org/pldoc/man?predicate=call_with_depth_limit/3) and [call\_with\_time\_limit/2](http://www.swi-prolog.org/pldoc/man?predicate=call_with_time_limit/2).

**callable**(*@Term*)

True if *Term* is bound to an atom or a compound term. This was intended as a type-test for arguments to [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1) and [call/2](http://www.swi-prolog.org/pldoc/man?predicate=call/2).. Note that callable only tests the *surface term*. Terms such as (22,true) are considered callable, but cause [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1) to raise a type error. Module-qualification of meta-argument (see [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1)) using **:/2** causes callable to succeed on any meta-argument.49 Consider the program and query below:

:- meta\_predicate p(0).

p(G) :- callable(G), call(G).

?- p(22).

ERROR: Type error: `callable' expected, found `22'

ERROR: In:

ERROR: [6] p(user:22)

**cancel\_halt**(*+Reason*)

If this predicate is called from a hook registered with [at\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=at_halt/1), halting Prolog is cancelled and an informational message is printed that includes *Reason*. This is used by the development tools to cancel halting the system if the editor has unsafed data and the user decides to cancel.

**catch**(*:Goal, +Catcher, :Recover*)

Behaves as [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1) if no exception is raised when executing *Goal*. If an exception is raised using[throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1) while *Goal* executes, and the *Goal* is the innermost goal for which *Catcher* unifies with the argument of [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1), all choice points generated by *Goal* are cut, the system backtracks to the start of [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) while preserving the thrown exception term, and *Recover* is called as in[call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1).

The overhead of calling a goal through [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) is comparable to [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1). Recovery from an exception is much slower, especially if the exception term is large due to the copying thereof.

**char\_code**(*?Atom, ?Code*)

Convert between character and character code for a single character.90

**char\_conversion**(*+CharIn, +CharOut*)

Define that term input (see [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3)) maps each character read as *CharIn* to the character*CharOut*. Character conversion is only executed if the Prolog flag [char\_conversion](http://www.swi-prolog.org/pldoc/man?section=flags#flag:char_conversion) is set totrue and not inside quoted atoms or strings. The initial table maps each character onto itself. See also [current\_char\_conversion/2](http://www.swi-prolog.org/pldoc/man?predicate=current_char_conversion/2).

**char\_type**(*?Char, ?Type*)

Tests or generates alternative *Type*s or *Char*s. The character types are inspired by the standard C <ctype.h> primitives.

**alnum**

*Char* is a letter (upper- or lowercase) or digit.

**alpha**

*Char* is a letter (upper- or lowercase).

**csym**

*Char* is a letter (upper- or lowercase), digit or the underscore (\_). These are valid C and Prolog symbol characters.

**csymf**

*Char* is a letter (upper- or lowercase) or the underscore (\_). These are valid first characters for C and Prolog symbols.

**ascii**

*Char* is a 7-bit ASCII character (0..127).

**white**

*Char* is a space or tab, i.e. white space inside a line.

**cntrl**

*Char* is an ASCII control character (0..31).

**digit**

*Char* is a digit.

**digit**(*Weight*)

*Char* is a digit with value *Weight*. I.e. char\_type(X, digit(6) yields *X* = '6'. Useful for parsing numbers.

**xdigit**(*Weight*)

*Char* is a hexadecimal digit with value *Weight*. I.e. char\_type(a, xdigit(X) yields *X* ='10'. Useful for parsing numbers.

**graph**

*Char* produces a visible mark on a page when printed. Note that the space is not included!

**lower**

*Char* is a lowercase letter.

**lower**(*Upper*)

*Char* is a lowercase version of *Upper*. Only true if *Char* is lowercase and *Upper*uppercase.

**to\_lower**(*Upper*)

*Char* is a lowercase version of *Upper*. For non-letters, or letter without case, *Char* and*Lower* are the same. See also [upcase\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=upcase_atom/2) and [downcase\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=downcase_atom/2).

**upper**

*Char* is an uppercase letter.

**upper**(*Lower*)

*Char* is an uppercase version of *Lower*. Only true if *Char* is uppercase and *Lower*lowercase.

**to\_upper**(*Lower*)

*Char* is an uppercase version of *Lower*. For non-letters, or letter without case, *Char* and*Lower* are the same. See also [upcase\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=upcase_atom/2) and [downcase\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=downcase_atom/2).

**punct**

*Char* is a punctuation character. This is a graph character that is not a letter or digit.

**space**

*Char* is some form of layout character (tab, vertical tab, newline, etc.).

**end\_of\_file**

*Char* is -1.

**end\_of\_line**

*Char* ends a line (ASCII: 10..13).

**newline**

*Char* is a newline character (10).

**period**

*Char* counts as the end of a sentence (.,!,?).

**quote**

*Char* is a quote character (", ', `).

**paren**(*Close*)

*Char* is an open parenthesis and *Close* is the corresponding close parenthesis.

**prolog\_var\_start**

*Char* can start a Prolog variable name.

**prolog\_atom\_start**

*Char* can start a unquoted Prolog atom that is not a symbol.

**prolog\_identifier\_continue**

*Char* can continue a Prolog variable name or atom.

**prolog\_prolog\_symbol**

*Char* is a Prolog symbol character. Sequences of Prolog symbol characters glue together to form an unquoted atom. Examples are =.., \=, etc.

**character\_count**(*+Stream, -Count*)

Unify *Count* with the current character index. For input streams this is the number of characters read since the open; for output streams this is the number of characters written. Counting starts at 0.

**chdir**(*+Path*)

Compatibility predicate. New code should use [working\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=working_directory/2).

**chr\_constraint**(*+Specifier*)

Every constraint used in CHR rules has to be declared with a [chr\_constraint/1](http://www.swi-prolog.org/pldoc/man?predicate=chr_constraint/1) declaration by the *constraint specifier*. For convenience multiple constraints may be declared at once with the same [chr\_constraint/1](http://www.swi-prolog.org/pldoc/man?predicate=chr_constraint/1) declaration followed by a comma-separated list of constraint specifiers.

A constraint specifier is, in its compact form, *F*/*A* where *F* and *A* are respectively the functor name and arity of the constraint, e.g.:

:- chr\_constraint foo/1.

:- chr\_constraint bar/2, baz/3.

In its extended form, a constraint specifier is *c*(*A\_1*, ... ,*A\_n*) where *c* is the constraint's functor, *n* its arity and the *A\_i* are argument specifiers. An argument specifier is a mode, optionally followed by a type. Example:

:- chr\_constraint get\_value(+,?).

:- chr\_constraint domain(?int, +list(int)),

alldifferent(?list(int)).

**chr\_show\_store**(*+Mod*)

Prints all suspended constraints of module *Mod* to the standard output. This predicate is automatically called by the SWI-Prolog top level at the end of each query for every CHR module currently loaded. The Prolog flag chr\_toplevel\_show\_store controls whether the top level shows the constraint stores. The value true enables it. Any other value disables it.

**chr\_trace**

Activate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) and [notrace/0](http://www.swi-prolog.org/pldoc/man?predicate=notrace/0).

**chr\_type**(*+TypeDeclaration*)

User-defined types are algebraic data types, similar to those in Haskell or the discriminated unions in Mercury. An algebraic data type is defined using [chr\_type/1](http://www.swi-prolog.org/pldoc/man?predicate=chr_type/1):

:- chr\_type type ---> body.

If the type term is a functor of arity zero (i.e. one having zero arguments), it names a monomorphic type. Otherwise, it names a polymorphic type; the arguments of the functor must be distinct type variables. The body term is defined as a sequence of constructor definitions separated by semi-colons.

Each constructor definition must be a functor whose arguments (if any) are types. Discriminated union definitions must be transparent: all type variables occurring in the body must also occur in the type.

Here are some examples of algebraic data type definitions:

:- chr\_type color ---> red ; blue ; yellow ; green.

:- chr\_type tree ---> empty ; leaf(int) ; branch(tree, tree).

:- chr\_type list(T) ---> [] ; [T | list(T)].

:- chr\_type pair(T1, T2) ---> (T1 - T2).

Each algebraic data type definition introduces a distinct type. Two algebraic data types that have the same bodies are considered to be distinct types (name equivalence).

Constructors may be overloaded among different types: there may be any number of constructors with a given name and arity, so long as they all have different types.

Aliases can be defined using ==. For example, if your program uses lists of lists of integers, you can define an alias as follows:

:- chr\_type lli == list(list(int)).

**chr\_notrace**

Deactivate the CHR tracer. By default the CHR tracer is activated and deactivated automatically by the Prolog predicates [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) and [notrace/0](http://www.swi-prolog.org/pldoc/man?predicate=notrace/0).

**chr\_leash**(*+Spec*)

Define the set of CHR ports on which the CHR tracer asks for user intervention (i.e. stops).*Spec* is either a list of ports as defined in [section 8.4.1](http://www.swi-prolog.org/pldoc/man?section=debugging) or a predefined `alias'. Defined aliases are: full to stop at all ports, none or off to never stop, and default to stop at the call, exit,fail, wake and apply ports. See also [leash/1](http://www.swi-prolog.org/pldoc/man?predicate=leash/1).

**chr\_option**(*+Option, +Value*)

It is possible to specify options that apply to all the CHR rules in the module. Options are specified with the chr\_option/2 declaration:

:- chr\_option(Option,Value).

and may appear in the file anywhere after the first constraints declaration.

Available options are:

**check\_guard\_bindings**

This option controls whether guards should be checked for (illegal) variable bindings or not. Possible values for this option are on to enable the checks, and off to disable the checks. If this option is on, any guard fails when it binds a variable that appears in the head of the rule. When the option is off (default), the behaviour of a binding in the guard is undefined.

**optimize**

This option controls the degree of optimization. Possible values are full to enable all available optimizations, and off (default) to disable all optimizations. The default is derived from the SWI-Prolog flag [optimise](http://www.swi-prolog.org/pldoc/man?section=flags#flag:optimise), where true is mapped to full. Therefore the command line option **-O** provides full CHR optimization. If optimization is enabled, debugging must be disabled.

**debug**

This option enables or disables the possibility to debug the CHR code. Possible values are on (default) and off. See [section 8.4](http://www.swi-prolog.org/pldoc/man?section=debugging) for more details on debugging. The default is derived from the Prolog flag [generate\_debug\_info](http://www.swi-prolog.org/pldoc/man?section=flags#flag:generate_debug_info), which is true by default. See **-nodebug**. If debugging is enabled, optimization must be disabled.

**clause**(*:Head, ?Body*)

True if *Head* can be unified with a clause head and *Body* with the corresponding clause body. Gives alternative clauses on backtracking. For facts, *Body* is unified with the atom *true*.

**clause**(*:Head, ?Body, ?Reference*)

Equivalent to [clause/2](http://www.swi-prolog.org/pldoc/man?predicate=clause/2), but unifies *Reference* with a unique reference to the clause (see also[assert/2](http://www.swi-prolog.org/pldoc/man?predicate=assert/2), [erase/1](http://www.swi-prolog.org/pldoc/man?predicate=erase/1)). If *Reference* is instantiated to a reference the clause's head and body will be unified with *Head* and *Body*.

**clause\_property**(*+ClauseRef, -Property*)

Queries properties of a clause. *ClauseRef* is a reference to a clause as produced by [clause/3](http://www.swi-prolog.org/pldoc/man?predicate=clause/3),[nth\_clause/3](http://www.swi-prolog.org/pldoc/man?predicate=nth_clause/3) or [prolog\_frame\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_frame_attribute/3). Unlike most other predicates that access clause references, [clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2) may be used to get information about erased clauses that have not yet been reclaimed. *Property* is one of the following:

**file**(*FileName*)

Unify *FileName* with the name of the file from which the clause is loaded. Fails if the clause was not created by loading a file (e.g., clauses added using [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1)). See alsosource.

**line\_count**(*LineNumber*)

Unify *LineNumber* with the line number of the clause. Fails if the clause is not associated to a file.

**size**(*SizeInBytes*)

True when *SizeInBytes* is the size that the clause uses in memory in bytes. The size required by a predicate also includes the predicate data record, a linked list of clauses, clause selection instructions and optionally one or more clause indexes.

**source**(*FileName*)

Unify *FileName* with the name of the source file that created the clause. This is the same as the file property, unless the file is loaded from a file that is textually included into source using [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1). In this scenario, file is the included file, while the sourceproperty refers to the *main* file.

**fact**

True if the clause has no body.

**erased**

True if the clause has been erased, but not yet reclaimed because it is referenced.

**predicate**(*PredicateIndicator*)

*PredicateIndicator* denotes the predicate to which this clause belongs. This is needed to obtain information on erased clauses because the usual way to obtain this information using [clause/3](http://www.swi-prolog.org/pldoc/man?predicate=clause/3) fails for erased clauses.

**module**(*Module*)

*Module* is the context module used to execute the body of the clause. For normal clauses, this is the same as the module in which the predicate is defined. However, if a clause is compiled with a module qualified *head*, the clause belongs to the predicate with the qualified head, while the body is executed in the context of the module in which the clause was defined.

**close**(*+Stream*)

Close the specified stream. If *Stream* is not open, an existence error is raised. See[stream\_pair/3](http://www.swi-prolog.org/pldoc/man?predicate=stream_pair/3) for the implications of closing a *stream pair*.

If the closed stream is the current input, output or error stream, the stream alias is bound to the initial standard I/O streams of the process. Calling [close/1](http://www.swi-prolog.org/pldoc/man?predicate=close/1) on the initial standard I/O streams of the process is a no-op for an input stream and flushes an output stream without closing it.74

**close**(*+Stream, +Options*)

Provides close(Stream, [force(true)]) as the only option. Called this way, any resource errors (such as write errors while flushing the output buffer) are ignored.

**close\_dde\_conversation**(*+Handle*)

Close the conversation associated with *Handle*. All opened conversations should be closed when they're no longer needed, although the system will close any that remain open on process termination.

**close\_shared\_object**(*+Handle*)

Detach the shared object identified by *Handle*.

**collation\_key**(*+Atom, -Key*)

Create a *Key* from *Atom* for locale-specific comparison. The key is defined such that if the key of atom *A* precedes the key of atom *B* in the standard order of terms, *A* is alphabetically smaller than *B* using the sort order of the current locale.

The predicate [collation\_key/2](http://www.swi-prolog.org/pldoc/man?predicate=collation_key/2) is used by [locale\_sort/2](http://www.swi-prolog.org/pldoc/man?predicate=locale_sort/2) from library(sort). Please examine the implementation of [locale\_sort/2](http://www.swi-prolog.org/pldoc/man?predicate=locale_sort/2) as an example of using this call.

The *Key* is an implementation-defined and generally unreadable string. On systems that do not support locale handling, *Key* is simply unified with *Atom*.

|  |  |
| --- | --- |
| **comment\_hook/3** | *(hook)* handle comments in sources |

**compare**(*?Order, @Term1, @Term2*)

Determine or test the *Order* between two terms in the standard order of terms. *Order* is one of<, > or =, with the obvious meaning.

**compile\_aux\_clauses**(*+Clauses*)

Compile clauses on behalf of [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2). This predicate compiles the argument clauses into static predicates, associating the predicates with the current file but avoids changing the notion of current predicate and therefore discontiguous warnings.

Note that in some cases multiple expansions of similar goals can share the same compiled auxiliary predicate. In such cases, the implementation of [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) can use[predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) using the property defined to test whether the predicate is already defined in the current context.

**compile\_predicates**(*:ListOfPredicateIndicators*)

Compile a list of specified dynamic predicates (see [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1) and [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1)) into normal static predicates. This call tells the Prolog environment the definition will not change anymore and further calls to [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1) or [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1) on the named predicates raise a permission error. This predicate is designed to deal with parts of the program that are generated at runtime but do not change during the remainder of the program execution.68

**compiling**

True if the system is compiling source files with the **-c** option or [qcompile/1](http://www.swi-prolog.org/pldoc/man?predicate=qcompile/1) into an intermediate code file. Can be used to perform conditional code optimisations in[term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2) (see also the **-O** option) or to omit execution of directives during compilation.

**compound**(*@Term*)

True if *Term* is bound to a compound term. See also [functor/3](http://www.swi-prolog.org/pldoc/man?predicate=functor/3) =../2, [compound\_name\_arity/3](http://www.swi-prolog.org/pldoc/man?predicate=compound_name_arity/3)and [compound\_name\_arguments/3](http://www.swi-prolog.org/pldoc/man?predicate=compound_name_arguments/3).

**compound\_name\_arity**(*?Compound, ?Name, ?Arity*)

Rationalized version of [functor/3](http://www.swi-prolog.org/pldoc/man?predicate=functor/3) that only works for compound terms and can examine and create compound terms with zero arguments (e.g, name(). See also[compound\_name\_arguments/3](http://www.swi-prolog.org/pldoc/man?predicate=compound_name_arguments/3).

**compound\_name\_arguments**(*?Compound, ?Name, ?Arguments*)

Rationalized version of [=../2](http://www.swi-prolog.org/pldoc/man?predicate=%3D../2) that can compose and decompose compound terms with zero arguments. See also [compound\_name\_arity/3](http://www.swi-prolog.org/pldoc/man?predicate=compound_name_arity/3).

**code\_type**(*?Code, ?Type*)

As [char\_type/2](http://www.swi-prolog.org/pldoc/man?predicate=char_type/2), but uses character codes rather than one-character atoms. Please note that both predicates are as flexible as possible. They handle either representation if the argument is instantiated and will instantiate only with an integer code or a one-character atom, depending of the version used. See also the Prolog flag [double\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:double_quotes), [atom\_chars/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2) and[atom\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_codes/2).

**consult**(*:File*)

Read *File* as a Prolog source file. Calls to [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1) may be abbreviated by just typing a number of filenames in a list. Examples:

|  |  |
| --- | --- |
| ?- consult(load). | % consult load or load.pl |
| ?- [library(lists)]. | % load library lists |
| ?- [user]. | % Type program on the terminal |

The predicate [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1) is equivalent to load\_files(File, []), except for handling the special file user, which reads clauses from the terminal. See also the stream(Input) option of[load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). Abbreviation using ?- [file1,file2]. does *not* work for the empty list ([]). This facility is implemented by defining the list as a predicate. Applications may only rely on using the list abbreviation at the Prolog toplevel and in directives.

**context\_module**(*-Module*)

Unify *Module* with the context module of the current goal. [context\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=context_module/1) itself is, of course, transparent.

**convert\_time***(+Stamp, -Y, -Mon, -Day, -Hour, -Min, -Sec, -MilliSec)*

Convert a time stamp, provided by [get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1), [time\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=time_file/2), etc. Year is unified with the year, Month with the month number (January is 1), *Day* with the day of the month (starting with 1),*Hour* with the hour of the day (0--23), Minute with the minute (0--59). Second with the second (0--59) and MilliSecond with the milliseconds (0--999). Note that the latter might not be accurate or might always be 0, depending on the timing capabilities of the system. See also[convert\_time/2](http://www.swi-prolog.org/pldoc/doc_for?object=backward_compatibility%3Aconvert_time/2).

**deprecated**

- Use [stamp\_date\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=stamp_date_time/3).

**convert\_time***(+Stamp, -String)*

Convert a time-stamp as obtained though [get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1) into a textual representation using the C-library function ctime(). The value is returned as a SWI-Prolog string object (see section 4.23). See also [convert\_time/8](http://www.swi-prolog.org/pldoc/doc_for?object=backward_compatibility%3Aconvert_time/8).

**deprecated**

- Use [format\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=format_time/3).

**copy\_stream\_data**(*+StreamIn, +StreamOut*)

Copy all (remaining) data from *StreamIn* to *StreamOut*.

**copy\_stream\_data**(*+StreamIn, +StreamOut, +Len*)

Copy *Len* codes from *StreamIn* to *StreamOut*. Note that the copy is done using the semantics of [get\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=get_code/2) and [put\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=put_code/2), taking care of possibly recoding that needs to take place between two text files. See [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars).

**copy\_predicate\_clauses**(*:From, :To*)

Copy all clauses of predicate *From* to *To*. The predicate *To* must be dynamic or undefined. If*To* is undefined, it is created as a dynamic predicate holding a copy of the clauses of *From*. If*To* is a dynamic predicate, the clauses of *From* are added (as in [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1)) to the clauses of *To*.*To* and *From* must have the same arity. Acts as if defined by the program below, but at a much better performance by avoiding decompilation and compilation.

copy\_predicate\_clauses(From, To) :-

head(From, MF:FromHead),

head(To, MT:ToHead),

FromHead =.. [\_|Args],

ToHead =.. [\_|Args],

forall(clause(MF:FromHead, Body),

assertz(MT:ToHead, Body)).

head(From, M:Head) :-

strip\_module(From, M, Name/Arity),

functor(Head, Name, Arity).

**copy\_term**(*+In, -Out*)

Create a version of *In* with renamed (fresh) variables and unify it to *Out*. Attributed variables (see [section 7.1](http://www.swi-prolog.org/pldoc/man?section=attvar)) have their attributes copied. The implementation of [copy\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/2) can deal with infinite trees (cyclic terms). As pure Prolog cannot distinguish a ground term from another ground term with exactly the same structure, ground sub-terms are *shared* between*In* and *Out*. Sharing ground terms does affect [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3). SWI-Prolog provides [duplicate\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=duplicate_term/2)to create a true copy of a term.

**copy\_term**(*+Term, -Copy, -Gs*)

Create a regular term *Copy* as a copy of *Term* (without any attributes), and a list *Gs* of goals that represents the attributes. The goal maplist(call,*Gs*) recreates the attributes for *Copy*. The nonterminal [attribute\_goals/3](http://www.swi-prolog.org/pldoc/man?predicate=attribute_goals/3), as defined in the modules the attributes stem from, is used to convert attributes to lists of goals.

This building block is used by the top level to report pending attributes in a portable and understandable fashion. This predicate is the preferred way to reason about and communicate terms with constraints.

**copy\_term\_nat**(*+Term, -Copy*)

As [copy\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/2). Attributes, however, are *not* copied but replaced by fresh variables.

**create\_prolog\_flag**(*+Key, +Value, +Options*)

Create a new Prolog flag. The ISO standard does not foresee creation of new flags, but many libraries introduce new flags. *Options* is a list of the options below. See also [user\_flags](http://www.swi-prolog.org/pldoc/man?section=flags#flag:user_flags).

**access**(*+Access*)

Define access rights for the flag. Values are read\_write and read\_only. The default isread\_write.

**type**(*+Atom*)

Define a type restriction. Possible values are boolean, atom, integer, float and term. The default is determined from the initial value. Note that term restricts the term to be ground.

**keep**(*+Boolean*)

If true, to not modify the flag if it already exists. Otherwise (default), this predicate behaves as [set\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=set_prolog_flag/2) if the flag already exists.

**current\_arithmetic\_function**(*?Head*)

True when *Head* is an evaluable function. For example:

?- current\_arithmetic\_function(sin(\_)).

true.

**current\_atom**(*-Atom*)

Successively unifies *Atom* with all atoms known to the system. Note that [current\_atom/1](http://www.swi-prolog.org/pldoc/man?predicate=current_atom/1)always succeeds if *Atom* is instantiated to an atom.

**current\_blob**(*?Blob, ?Type*)

Examine the type or enumerate blobs of the given *Type*. Typed blobs are supported through the foreign language interface for storing arbitrary BLOBs (Binary Large Object) or handles to external entities. See [section 10.4.7](http://www.swi-prolog.org/pldoc/man?section=foreigninclude) for details.

**current\_char\_conversion**(*?CharIn, ?CharOut*)

Queries the current character conversion table. See [char\_conversion/2](http://www.swi-prolog.org/pldoc/man?predicate=char_conversion/2) for details.

**current\_flag**(*-FlagKey*)

Successively unifies *FlagKey* with all keys used for flags (see [flag/3](http://www.swi-prolog.org/pldoc/man?predicate=flag/3)).

**current\_foreign\_library**(*?File, ?Public*)

Query currently loaded shared libraries.

**current\_format\_predicate**(*?Code, ?:Head*)

True when ~*Code* is handled by the user-defined predicate specified by *Head*.

**current\_functor**(*?Name, ?Arity*)

Successively unifies *Name* with the name and *Arity* with the arity of functors known to the system.

**current\_input**(*-Stream*)

Get the current input stream. Useful for getting access to the status predicates associated with streams.

**current\_key**(*-Key*)

Successively unifies *Key* with all keys used for records (see [recorda/3](http://www.swi-prolog.org/pldoc/man?predicate=recorda/3), etc.).

**current\_locale**(*-Locale*)

True when *Locale* is the locale of the calling thread.

**current\_module**(*?Module*)

True if *Module* is a currently defined module. This predicate enumerates all modules, whether loaded from a file or created dynamically. Note that modules cannot be destroyed in the current version of SWI-Prolog.

**current\_op**(*?Precedence, ?Type, ?:Name*)

True if *Name* is currently defined as an operator of type *Type* with precedence *Precedence*. See also [op/3](http://www.swi-prolog.org/pldoc/man?predicate=op/3).

**current\_output**(*-Stream*)

Get the current output stream.

**current\_predicate**(*:PredicateIndicator*)

True if *PredicateIndicator* is a currently defined predicate. A predicate is considered defined if it exists in the specified module, is imported into the module or is defined in one of the modules from which the predicate will be imported if it is called (see [section 6.9](http://www.swi-prolog.org/pldoc/man?section=importmodule)). Note that[current\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=current_predicate/1) does *not* succeed for predicates that can be *autoloaded*. See also[current\_predicate/2](http://www.swi-prolog.org/pldoc/man?predicate=current_predicate/2) and [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2).

If *PredicateIndicator* is not fully specified, the predicate only generates values that are defined in or already imported into the target module. Generating all callable predicates therefore requires enumerating modules using [current\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=current_module/1). Generating predicates callable in a given module requires enumerating the import modules using [import\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=import_module/2) and the autoloadable predicates using the [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) autoload.

**current\_predicate**(*?Name, :Head*)

Classical pre-ISO implementation of [current\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=current_predicate/1), where the predicate is represented by the head term. The advantage is that this can be used for checking the existence of a predicate before calling it without the need for [functor/3](http://www.swi-prolog.org/pldoc/man?predicate=functor/3):

call\_if\_exists(G) :-

current\_predicate(\_, G),

call(G).

Because of this intended usage, [current\_predicate/2](http://www.swi-prolog.org/pldoc/man?predicate=current_predicate/2) also succeeds if the predicate can be autoloaded. Unfortunately, checking the autoloader makes this predicate relatively slow, in particular because a failed lookup of the autoloader will cause the autoloader to verify that its index is up-to-date.

**current\_signal**(*?Name, ?Id, ?Handler*)

Enumerate the currently defined signal handling. *Name* is the signal name, *Id* is the numerical identifier and *Handler* is the currently defined handler (see [on\_signal/3](http://www.swi-prolog.org/pldoc/man?predicate=on_signal/3)).

**current\_stream**(*?Object, ?Mode, ?Stream*)

The predicate [current\_stream/3](http://www.swi-prolog.org/pldoc/man?predicate=current_stream/3) is used to access the status of a stream as well as to generate all open streams. *Object* is the name of the file opened if the stream refers to an open file, an integer file descriptor if the stream encapsulates an operating system stream, or the atom []if the stream refers to some other object. *Mode* is one of read or write.

**cyclic\_term**(*@Term*)

True if *Term* contains cycles, i.e. is an infinite term. See also [acyclic\_term/1](http://www.swi-prolog.org/pldoc/man?predicate=acyclic_term/1) and [section 2.16](http://www.swi-prolog.org/pldoc/man?section=cyclic).50

**day\_of\_the\_week**(*+Date,-DayOfTheWeek*)

Computes the day of the week for a given date. *Date* = date(*Year*,*Month*,*Day*). Days of the week are numbered from one to seven: Monday = 1, Tuesday = 2, ... , Sunday = 7.

**date\_time\_stamp**(*+DateTime, -TimeStamp*)

Compute the timestamp from a date/9 term. Values for month, day, hour, minute or second need not be normalized. This flexibility allows for easy computation of the time at any given number of these units from a given timestamp. Normalization can be achieved following this call with [stamp\_date\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=stamp_date_time/3). This example computes the date 200 days after 2006-7-14:

?- date\_time\_stamp(date(2006,7,214,0,0,0,0,-,-), Stamp),

stamp\_date\_time(Stamp, D, 0),

date\_time\_value(date, D, Date).

Date = date(2007, 1, 30)

When computing a time stamp from a local time specification, the UTC offset (arg 7), TZ (arg 8) and DST (arg 9) argument may be left unbound and are unified with the proper information. The example below, executed in Amsterdam, illustrates this behaviour. On the 25th of March at 01:00, DST does not apply. At 02.00, the clock is advanced by one hour and thus both 02:00 and 03:00 represent the same time stamp.

1 ?- date\_time\_stamp(date(2012,3,25,1,0,0,UTCOff,TZ,DST),

Stamp).

UTCOff = -3600,

TZ = 'CET',

DST = false,

Stamp = 1332633600.0.

2 ?- date\_time\_stamp(date(2012,3,25,2,0,0,UTCOff,TZ,DST),

Stamp).

UTCOff = -7200,

TZ = 'CEST',

DST = true,

Stamp = 1332637200.0.

3 ?- date\_time\_stamp(date(2012,3,25,3,0,0,UTCOff,TZ,DST),

Stamp).

UTCOff = -7200,

TZ = 'CEST',

DST = true,

Stamp = 1332637200.0.

Note that DST and offset calculation are based on the POSIX function mktime(). If mktime() returns an error, a representation\_error dst is generated.

**date\_time\_value**(*?Key, +DateTime, ?Value*)

Extract values from a date/9 term. Provided keys are:

|  |  |
| --- | --- |
| **key** | **value** |
| year | Calendar year as an integer |
| month | Calendar month as an integer 1..12 |
| day | Calendar day as an integer 1..31 |
| hour | Clock hour as an integer 0..23 |
| minute | Clock minute as an integer 0..59 |
| second | Clock second as a float 0.0..60.0 |
| utc\_offset | Offset to UTC in seconds (positive is west) |
| time\_zone | Name of timezone; fails if unknown |
| daylight\_saving | Bool daylight\_savingtrue) if dst is in effect |
| date | Term date(Y,M,D) |
| time | Term time(H,M,S) |

**dcg\_translate\_rule**(*+In, -Out*)

This predicate performs the translation of a term Head-->Body into a normal Prolog clause. Normally this functionality should be accessed using [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2).

# [dcg\_translate\_rule/4](http://www.swi-prolog.org/pldoc/consulting.html#dcg_translate_rule/4)

**dde\_current\_connection**(*-Service, -Topic*)

Find currently open conversations.

**dde\_current\_service**(*-Service, -Topic*)

Find currently registered services and the topics served on them.

**dde\_execute**(*+Handle, +Command*)

Request the DDE server to execute the given command string. Succeeds if the command could be executed and fails with an error message otherwise.

**dde\_register\_service**(*+Template, +Goal*)

Register a server to handle DDE request or DDE execute requests from other applications. To register a service for a DDE request, *Template* is of the form:

+Service(+Topic, +Item, +Value)

*Service* is the name of the DDE service provided (like **progman** in the client example above).*Topic* is either an atom, indicating *Goal* only handles requests on this topic, or a variable that also appears in *Goal*. *Item* and *Value* are variables that also appear in *Goal*. *Item* represents the request data as a Prolog atom.129

The example below registers the Prolog [current\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=current_prolog_flag/2) predicate to be accessible from other applications. The request may be given from the same Prolog as well as from another application.

?- dde\_register\_service(prolog(current\_prolog\_flag, F, V),

current\_prolog\_flag(F, V)).

?- open\_dde\_conversation(prolog, current\_prolog\_flag, Handle),

dde\_request(Handle, home, Home),

close\_dde\_conversation(Handle).

Home = '/usr/local/lib/pl-2.0.6/'

Handling DDE execute requests is very similar. In this case the template is of the form:

+Service(+Topic, +Item)

Passing a *Value* argument is not needed as execute requests either succeed or fail. If *Goal*fails, a `not processed' is passed back to the caller of the DDE request.

**dde\_request**(*+Handle, +Item, -Value*)

Request a value from the server. *Item* is an atom that identifies the requested data, and *Value*will be a string (CF\_TEXT data in DDE parlance) representing that data, if the request is successful.

|  |  |
| --- | --- |
| **dde\_poke/3** | Win32: POKE operation on DDE server |

**dde\_unregister\_service**(*+Service*)

Stop responding to *Service*. If Prolog is halted, it will automatically call this on all open services.

**debug**

Start debugger. In debug mode, Prolog stops at spy and trace points, disables last-call optimisation and aggressive destruction of choice points to make debugging information accessible. Implemented by the Prolog flag [debug](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debug).

Note that the min\_free parameter of all stacks is enlarged to 8 K cells if debugging is switched off in order to avoid excessive GC. GC complicates tracing because it renames the *\_G<NNN>*variables and replaces unreachable variables with the atom <garbage\_collected>. Calling[nodebug/0](http://www.swi-prolog.org/pldoc/man?predicate=nodebug/0) does *not* reset the initial free-margin because several parts of the top level and debugger disable debugging of system code regions. See also [set\_prolog\_stack/2](http://www.swi-prolog.org/pldoc/man?predicate=set_prolog_stack/2).

**debug**(*+Topic*)

*[det]***nodebug**(*+Topic*)

Add/remove a topic from being printed. nodebug(\_) removes all topics. Gives a warning if the topic is not defined unless it is used from a directive. The latter allows placing debug topics at the start of a (load-)file without warnings.

For [debug/1](http://www.swi-prolog.org/pldoc/man?predicate=debug/1), *Topic* can be a term *Topic* *>* Out, where Out is either a stream or stream-alias or a filename (atom). This redirects debug information on this topic to the given output.

**debug**(*+Topic, +Format, :Args*)

*Format* a message if debug topic is enabled. Similar to [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3) to user\_error, but only prints if *Topic* is activated through [debug/1](http://www.swi-prolog.org/pldoc/man?predicate=debug/1). *Args* is a meta-argument to deal with goal for the @-command. Output is first handed to the hook [**prolog:debug\_print\_hook/3**](http://www.swi-prolog.org/pldoc/debug.html#prolog:debug_print_hook/3). If this fails,*Format*+*Args* is translated to text using the message-translation (see [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2)) for the term debug(Format, Args) and then printed to every matching destination (controlled by[debug/1](http://www.swi-prolog.org/pldoc/man?predicate=debug/1)) using [print\_message\_lines/3](http://www.swi-prolog.org/pldoc/man?predicate=print_message_lines/3).

The message is preceded by '% ' and terminated with a newline.

**See also**

[format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3).

|  |  |
| --- | --- |
| **debug\_control\_hook/1** | *(hook)* Extend [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1), etc. |

**debugging**

Print debug status and spy points on current output stream. See also the Prolog flag [debug](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debug).

**debugging**(*+Topic*)

*[nondet]***debugging**(*-Topic*)

*[nondet]***debugging**(*?Topic, ?Bool*)

Examine debug topics. The form debugging(+Topic) may be used to perform more complex debugging tasks. A typical usage skeleton is:

( debugging(mytopic)

-> <perform debugging actions>

; true

),

...

The other two calls are intended to examine existing and enabled debugging tokens and are typically not used in user programs.

**default\_module**(*+Module, -Default*)

True if predicates and operators in *Default* are visible in *Module*. Modules are returned in the same search order used for predicates and operators. That is, *Default* is first unified with*Module*, followed by the depth-first transitive closure of [import\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=import_module/2).

**del\_attr**(*+Var, +Module*)

Delete the named attribute. If *Var* loses its last attribute it is transformed back into a traditional Prolog variable. If *Module* is not an atom, a type error is raised. In all other cases this predicate succeeds regardless of whether or not the named attribute is present.

**del\_attrs**(*+Var*)

If *Var* is an attributed variable, delete *all* its attributes. In all other cases, this predicate succeeds without side-effects.

**del\_dict**(*+Key, +DictIn, ?Value, -DictOut*)

True when *Key*-*Value* is in *DictIn* and *DictOut* contains all associations of *DictIn* except for *Key*.

**delete\_directory**(*+Directory*)

Delete directory (folder) from the filesystem. Raises an exception on failure. Please note that in general it will not be possible to delete a non-empty directory.

**delete\_file**(*+File*)

Remove *File* from the file system.

**delete\_import\_module**(*+Module, +Import*)

Delete *Import* from the list of import modules for *Module*. Fails silently if *Import* is not in the list.

**deterministic**(*-Boolean*)

Unifies its argument with true if no choice point exists that is more recent than the entry of the clause in which it appears. There are few realistic situations for using this predicate. It is used by the [prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0) top level to check whether Prolog should prompt the user for alternatives. Similar results can be achieved in a more portable fashion using [call\_cleanup/2](http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2).

**dif**(*@A, @B*)

The [dif/2](http://www.swi-prolog.org/pldoc/man?predicate=dif/2) predicate provides a constraint stating that *A* and *B* are different terms. If *A* and *B*can never unify, [dif/2](http://www.swi-prolog.org/pldoc/man?predicate=dif/2) succeeds deterministically. If *A* and *B* are identical it fails immediately, and finally, if *A* and *B* can unify, goals are delayed that prevent *A* and *B* to become equal. The[dif/2](http://www.swi-prolog.org/pldoc/man?predicate=dif/2) predicate behaves as if defined by dif(X, Y) :- when(?=(X, Y), X \== Y). See also [?=/2](http://www.swi-prolog.org/pldoc/man?predicate=?%3D/2). The implementation can deal with cyclic terms.

The [dif/2](http://www.swi-prolog.org/pldoc/man?predicate=dif/2) predicate is realised using attributed variables associated with the module dif. It is defined in the autoload library library(dif).

**directory\_files**(*+Directory, -Entries*)

Unify *Entries* with a list of entries in *Directory*. Each member of *Entries* is an atom denoting an entry relative to *Directory*. *Entries* contains all entries, including hidden files and, if supplied by the OS, the special entries . and ... See also [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2).123

**discontiguous** *:PredicateIndicator, ...*

Informs the system that the clauses of the specified predicate(s) might not be together in the source file. See also [style\_check/1](http://www.swi-prolog.org/pldoc/man?predicate=style_check/1).

**divmod**(*+Dividend, +Divisor, -Quotient, -Remainder*)

This predicate is a shorthand for computing both the *Quotient* and *Remainder* of two integers in a single operation. This allows for exploiting the fact that the low level implementation for computing the quotient also produces the remainder. Timing confirms that this predicate is almost twice as fast as performing the steps independently. Semantically, [divmod/4](http://www.swi-prolog.org/pldoc/man?predicate=divmod/4) is defined as below.

divmod(Dividend, Divisor, Quotient, Remainder) :-

Quotient is Dividend div Divisor,

Remainder is Dividend mod Divisor.

Note that this predicate is only available if SWI-Prolog is compiled with unbounded integer support. This is the case for all packaged versions.

**downcase\_atom**(*+AnyCase, -LowerCase*)

Converts the characters of *AnyCase* into lowercase as [char\_type/2](http://www.swi-prolog.org/pldoc/man?predicate=char_type/2) does (i.e. based on the defined *locale* if Prolog provides locale support on the hosting platform) and unifies the lowercase atom with *LowerCase*.

**duplicate\_term**(*+In, -Out*)

Version of [copy\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/2) that also copies ground terms and therefore ensures that destructive modification using [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) does not affect the copy. See also [nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2), [nb\_linkval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_linkval/2),[nb\_setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_setarg/3) and [nb\_linkarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_linkarg/3).

**dwim\_match**(*+Atom1, +Atom2*)

True if *Atom1* matches *Atom2* in the `Do What I Mean' sense. Both *Atom1* and *Atom2* may also be integers or floats. The two atoms match if:

* They are identical
* They differ by one character (spy *==* spu)
* One character is inserted/deleted (debug *==* deug)
* Two characters are transposed (trace *==* tarce)
* `Sub-words' are glued differently (existsfile *==* existsFile *==* exists\_file)
* Two adjacent sub-words are transposed (existsFile *==* fileExists)

**dwim\_match**(*+Atom1, +Atom2, -Difference*)

Equivalent to [dwim\_match/2](http://www.swi-prolog.org/pldoc/man?predicate=dwim_match/2), but unifies *Difference* with an atom identifying the difference between *Atom1* and *Atom2*. The return values are (in the same order as above): equal,mismatched\_char, inserted\_char, transposed\_char, separated and transposed\_word.

**dwim\_predicate**(*+Term, -Dwim*)

`Do What I Mean' (`dwim') support predicate. *Term* is a term, whose name and arity are used as a predicate specification. *Dwim* is instantiated with the most general term built from *Name*and the arity of a defined predicate that matches the predicate specified by *Term* in the `Do What I Mean' sense. See [dwim\_match/2](http://www.swi-prolog.org/pldoc/man?predicate=dwim_match/2) for `Do What I Mean' string matching. Internal system predicates are not generated, unless the access level is system (see [access\_level](http://www.swi-prolog.org/pldoc/man?section=flags#flag:access_level)). Backtracking provides all alternative matches.

**dynamic** *:PredicateIndicator, ...*

Informs the interpreter that the definition of the predicate(s) may change during execution (using [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1) and/or [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1)). In the multithreaded version, the clauses of dynamic predicates are shared between the threads. The directive [thread\_local/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_local/1) provides an alternative where each thread has its own clause list for the predicate. Dynamic predicates can be turned into static ones using [compile\_predicates/1](http://www.swi-prolog.org/pldoc/man?predicate=compile_predicates/1).

**edit**

Edit the `default' file using [edit/1](http://www.swi-prolog.org/pldoc/man?predicate=edit/1). The default file is the file loaded with the command line option **-s** or, in Windows, the file loaded by double-clicking from the Windows shell.

**edit**(*+Specification*)

First, exploit prolog\_edit:locate/3 to translate *Specification* into a list of *Locations*. If there is more than one `hit', the user is asked to select from the locations found. Finally, prolog\_edit:edit\_source/1 is used to invoke the user's preferred editor. Typically, [edit/1](http://www.swi-prolog.org/pldoc/man?predicate=edit/1) can be handed the name of a predicate, module, basename of a file, XPCE class, XPCE method, etc.

**elif**(*:Goal*)

Equivalent to :- else. :-if(Goal). ... :- endif. In a sequence as below, the section below the first matching elif is processed. If no test succeeds, the else branch is processed.

:- if(test1).

section\_1.

:- elif(test2).

section\_2.

:- elif(test3).

section\_3.

:- else.

section\_else.

:- endif.

**else**

Start `else' branch.

**empty\_assoc**(*?Assoc*)

*Assoc* is unified with an empty association list.

**empty\_nb\_set**(*?Set*)

True if *Set* is a non-backtrackable empty set.

**encoding**(*+Encoding*)

This directive can appear anywhere in a source file to define how characters are encoded in the remainder of the file. It can be used in files that are encoded with a superset of US-ASCII, currently UTF-8 and ISO Latin-1. See also [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars).

**endif**

End of conditional compilation.

**ensure\_loaded**(*:File*)

If the file is not already loaded, this is equivalent to [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1). Otherwise, if the file defines a module, import all public predicates. Finally, if the file is already loaded, is not a module file, and the context module is not the global user module, [ensure\_loaded/1](http://www.swi-prolog.org/pldoc/man?predicate=ensure_loaded/1) will call [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1).

With this semantics, we hope to get as close as possible to the clear semantics without the presence of a module system. Applications using modules should consider usinguse\_module/[1,2].

Equivalent to load\_files(Files, [if(not\_loaded)]).41

**erase**(*+Reference*)

Erase a record or clause from the database. *Reference* is a db-reference returned by[recorda/3](http://www.swi-prolog.org/pldoc/man?predicate=recorda/3), [recordz/3](http://www.swi-prolog.org/pldoc/man?predicate=recordz/3) or [recorded/3](http://www.swi-prolog.org/pldoc/man?predicate=recorded/3), [clause/3](http://www.swi-prolog.org/pldoc/man?predicate=clause/3), [assert/2](http://www.swi-prolog.org/pldoc/man?predicate=assert/2), [asserta/2](http://www.swi-prolog.org/pldoc/man?predicate=asserta/2) or [assertz/2](http://www.swi-prolog.org/pldoc/man?predicate=assertz/2). Fail silently if the referenced object no longer exists.

**eval\_license**

Evaluate the license conditions of all loaded components. If the system contains one or more components that are licenced under GPL-like restrictions the system indicates this program may only be distributed under the GPL license as well as which components prohibit the use of other license conditions.

**exception**(*+Exception, +Context, -Action*)

Dynamic predicate, normally not defined. Called by the Prolog system on run-time exceptions that can be repaired `just-in-time'. The values for *Exception* are described below. See also[catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) and [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1).

If this hook predicate succeeds it must instantiate the *Action* argument to the atom fail to make the operation fail silently, retry to tell Prolog to retry the operation or error to make the system generate an exception. The action retry only makes sense if this hook modified the environment such that the operation can now succeed without error.

**undefined\_predicate**

*Context* is instantiated to a predicate indicator ([module]:<*name*>/<*arity*>). If the predicate fails, Prolog will generate an existence\_error exception. The hook is intended to implement alternatives to the built-in autoloader, such as autoloading code from a database. Do *not* use this hook to suppress existence errors on predicates. See also [unknown](http://www.swi-prolog.org/pldoc/man?section=flags#flag:unknown) and [section 2.13](http://www.swi-prolog.org/pldoc/man?section=autoload).

**undefined\_global\_variable**

*Context* is instantiated to the name of the missing global variable. The hook must call[nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2) or [b\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=b_setval/2) before returning with the action retry.

**exists\_directory**(*+Directory*)

True if *Directory* exists and is a directory. This does not imply the user has read, search or write permission for the directory.

**exists\_file**(*+File*)

True if *File* exists and is a regular file. This does not imply the user has read and/or write permission for the file. This is the same as access\_file(File, exist).

**exists\_source**(*+Spec*)

Is true if *Spec* exists as a Prolog source. *Spec* uses the same conventions as [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). Fails without error if *Spec* cannot be found.

**expand\_answer**(*+Bindings, -ExpandedBindings*)

Hook in module user, normally not defined. Expand the result of a successfully executed top-level query. *Bindings* is the query *<Name>=<Value>* binding list from the query.*ExpandedBindings* must be unified with the bindings the top level should print.

**expand\_file\_name**(*+WildCard, -List*)

Unify *List* with a sorted list of files or directories matching *WildCard*. The normal Unix wildcard constructs `?', `\*', `[ ... ]' and `{...}' are recognised. The interpretation of `{...}' is slightly different from the C shell (csh(1)). The comma-separated argument can be arbitrary patterns, including `{...}' patterns. The empty pattern is legal as well: `{.pl,}' matches either `.pl' or the empty string.

If the pattern contains wildcard characters, only existing files and directories are returned. Expanding a `pattern' without wildcard characters returns the argument, regardless of whether or not it exists.

Before expanding wildcards, the construct $*var* is expanded to the value of the environment variable *var*, and a possible leading ~ character is expanded to the user's home directory.124

**expand\_file\_search\_path**(*+Spec, -Path*)

Unifies *Path* with all possible expansions of the filename specification *Spec*. See also[absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3).

**expand\_goal**(*+Goal1, -Goal2*)

This predicate is normally called by the compiler to perform preprocessing using[goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2). The predicate computes a fixed-point by applying transformations until there are no more changes. If optimisation is enabled (see **-O** and [optimise](http://www.swi-prolog.org/pldoc/man?section=flags#flag:optimise)), [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2)simplifies the result by removing unneeded calls to [true/0](http://www.swi-prolog.org/pldoc/man?predicate=true/0) and [fail/0](http://www.swi-prolog.org/pldoc/man?predicate=fail/0) as well as unreachable branches.

**expand\_goal**(*+Goal1, ?Layout1, -Goal2, -Layout2*)

**goal\_expansion**(*+Goal1, ?Layout1, -Goal2, -Layout2*)

**expand\_term**(*+Term1, ?Layout1, -Term2, -Layout2*)

**term\_expansion**(*+Term1, ?Layout1, -Term2, -Layout2*)

**dcg\_translate\_rule**(*+In, ?LayoutIn, -Out, -LayoutOut*)

These versions are called *before* their 2-argument counterparts. The input layout term is either a variable (if no layout information is available) or a term carrying detailed layout information as returned by the subterm\_positions of [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2).

**expand\_query**(*+Query, -Expanded, +Bindings, -ExpandedBindings*)

Hook in module user, normally not defined. *Query* and *Bindings* represents the query read from the user and the names of the free variables as obtained using [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3). If this predicate succeeds, it should bind *Expanded* and *ExpandedBindings* to the query and bindings to be executed by the top level. This predicate is used by the top level ([prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0)). See also [expand\_answer/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_answer/2) and [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2).

**expand\_term**(*+Term1, -Term2*)

This predicate is normally called by the compiler on terms read from the input to perform preprocessing. It consists of four steps, where each step processes the output of the previous step.

Test conditional compilation directives and translate all input to [] if we are in a `false branch' of the conditional compilation. See [section 4.3.1.2](http://www.swi-prolog.org/pldoc/man?section=consulting).

Call [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2). This predicate is first tried in the module that is being compiled and then in the module user.

Call DCG expansion ([dcg\_translate\_rule/2](http://www.swi-prolog.org/pldoc/man?predicate=dcg_translate_rule/2)).

Call [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2) on each body term that appears in the output of the previous steps.

# [expand\_term/4](http://www.swi-prolog.org/pldoc/consulting.html#expand_term/4) Not Found

**expects\_dialect**(*+Dialect*)

This directive states that the code following the directive is written for the given Prolog *Dialect*. See also [dialect](http://www.swi-prolog.org/pldoc/man?section=flags#flag:dialect). The declaration holds until the end of the file in which it appears. The current dialect is available using [prolog\_load\_context/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_load_context/2).

The exact behaviour of this predicate is still subject to discussion. Of course, if *Dialect*matches the running dialect the directive has no effect. Otherwise we check for the existence of library(dialect/Dialect) and load it if the file is found. Currently, this file has this functionality:

Define system predicates of the requested dialect we do not have.

Apply [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) rules that map conflicting predicates to versions emulating the requested dialect. These expansion rules reside in the dialect compatibility module, but are applied if prolog\_load\_context(dialect, Dialect) is active.

Modify the search path for library directories, putting libraries compatible with the target dialect before the native libraries.

Setup support for the default filename extension of the dialect.

**explain**(*+ToExplain*)

Give an explanation on the given `object'. The argument may be any Prolog data object. If the argument is an atom, a term of the form *Name/Arity* or a term of the form *Module:Name/Arity*,[explain/1](http://www.swi-prolog.org/pldoc/man?predicate=explain/1) describes the predicate as well as possible references to it. See also [gxref/0](http://www.swi-prolog.org/pldoc/man?predicate=gxref/0).

**explain**(*+ToExplain, -Explanation*)

Unify *Explanation* with an explanation for *ToExplain*. Backtracking yields further explanations.

**export**(*+PredicateIndicator, ...*)

Add predicates to the public list of the context module. This implies the predicate will be imported into another module if this module is imported with use\_module/[1,2]. Note that predicates are normally exported using the directive [module/2](http://www.swi-prolog.org/pldoc/man?predicate=module/2). [export/1](http://www.swi-prolog.org/pldoc/man?predicate=export/1) is meant to handle export from dynamically created modules.

**fail**

Always fail. The predicate [fail/0](http://www.swi-prolog.org/pldoc/man?predicate=fail/0) is translated into a single virtual machine instruction.

**false**

Same as fail, but the name has a more declarative connotation.

**current\_prolog\_flag**(*?Key, -Value*)

The predicate [current\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=current_prolog_flag/2) defines an interface to installation features: options compiled in, version, home, etc. With both arguments unbound, it will generate all defined Prolog flags. With `Key' instantiated, it unifies the value of the Prolog flag. Flag values are typed. Flags marked as bool can have the values true or false. Some Prolog flags are not defined in all versions, which is normally indicated in the documentation below as *``if present and true''*. A boolean Prolog flag is true iff the Prolog flag is present **and** the *Value* is the atom true. Tests for such flags should be written as below:

( current\_prolog\_flag(windows, true)

-> <Do MS-Windows things>

; <Do normal things>

)

Some Prolog flags are scoped to a source file. This implies that if they are set using a directive inside a file, the flag value encountered when loading of the file started is restored when loading of the file is completed. Currently, the following flags are scoped to the source file:[generate\_debug\_info](http://www.swi-prolog.org/pldoc/man?section=flags#flag:generate_debug_info) and [optimise](http://www.swi-prolog.org/pldoc/man?section=flags#flag:optimise).

A new thread (see [section 9](http://www.swi-prolog.org/pldoc/man?section=threads)) *copies* all flags from the thread that created the new thread (its*parent*).14 As a consequence, modifying a flag inside a thread does not affect other threads.

**access\_level**(*atom*, changeable)

This flag defines a normal `user' view (user, default) or a `system' view. In system view all system code is fully accessible as if it was normal user code. In user view, certain operations are not permitted and some details are kept invisible. We leave the exact consequences undefined, but, for example, system code can be traced using system access and system predicates can be redefined.

**address\_bits**(*integer*)

Address size of the hosting machine. Typically 32 or 64. Except for the maximum stack limit, this has few implications to the user. See also the Prolog flag [arch](http://www.swi-prolog.org/pldoc/man?section=flags#flag:arch).

**agc\_margin**(*integer*, changeable)

If this amount of atoms possible garbage atoms exist perform atom garbage collection at the first opportunity. Initial value is 10,000. May be changed. A value of 0 (zero) disables atom garbage collection. See also [PL\_register\_atom()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_register_atom).15

**apple**(*bool*)

If present and true, the operating system is MacOSX. Defined if the C compiler used to compile this version of SWI-Prolog defines \_\_APPLE\_\_. Note that the [unix](http://www.swi-prolog.org/pldoc/man?section=flags#flag:unix) is also defined for MacOSX.

**allow\_dot\_in\_atom**(*bool*, changeable)

If true (default false), dots may be embedded into atoms that are not quoted and start with a letter. The embedded dot *must* be followed by an identifier continuation character (i.e., letter, digit or underscore). The dot is allowed in identifiers in many languages, which can make this a useful flag for defining DSLs. Note that this conflicts with cascading functional notation. For example, Post.meta.author is read as .(Post, 'meta.author' if this flag is set to true.

**allow\_variable\_name\_as\_functor**(*bool*, changeable)

If true (default is false), Functor(arg) is read as if it were written 'Functor'(arg). Some applications use the Prolog [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) predicate for reading an application-defined script language. In these cases, it is often difficult to explain to non-Prolog users of the application that constants and functions can only start with a lowercase letter. Variables can be turned into atoms starting with an uppercase atom by calling [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) using the option variable\_names and binding the variables to their name. Using this feature, F(x) can be turned into valid syntax for such script languages. Suggested by Robert van Engelen. SWI-Prolog specific.

**argv**(*list*, changeable)

List is a list of atoms representing the application command line arguments. Application command line arguments are those that have *not* been processed by Prolog during its initialization. Note that Prolog's argument processing stops at -- or the first non-option argument. See also [os\_argv](http://www.swi-prolog.org/pldoc/man?section=flags#flag:os_argv).16

**arch**(*atom*)

Identifier for the hardware and operating system SWI-Prolog is running on. Used to select foreign files for the right architecture. See also [section 10.2.3](http://www.swi-prolog.org/pldoc/man?section=foreignlink) and[file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2).

**associated\_file**(*atom*)

Set if Prolog was started with a prolog file as argument. Used by e.g., [edit/0](http://www.swi-prolog.org/pldoc/man?predicate=edit/0) to edit the initial file.

**autoload**(*bool*, changeable)

If true (default) autoloading of library functions is enabled.

**back\_quotes**(*codes,chars,string,symbol\_char*, changeable)

Defines the term-representation for back-quoted material. The default is codes. If **--traditional** is given, the default is symbol\_char, which allows using ` in operators composed of symbols.17. See also [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings).

**bounded**(*bool*)

ISO Prolog flag. If true, integer representation is bound by [min\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:min_integer) and[max\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:max_integer). If false integers can be arbitrarily large and the [min\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:min_integer) and[max\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:max_integer) are not present. See [section 4.26.2.1](http://www.swi-prolog.org/pldoc/man?section=arith).

**break\_level**(*integer*)

Current break-level. The initial top level (started with **-t**) has value 0. See [break/0](http://www.swi-prolog.org/pldoc/man?predicate=break/0). This flag is absent from threads that are not running a top-level loop.

**c\_cc**(*atom*, changeable)

Name of the C compiler used to compile SWI-Prolog. Normally either gcc or cc. See[section 10.5](http://www.swi-prolog.org/pldoc/man?section=plld).

**c\_cflags**(*atom*, changeable)

CFLAGS used to compile SWI-Prolog. See [section 10.5](http://www.swi-prolog.org/pldoc/man?section=plld).

**c\_ldflags**(*atom*, changeable)

LDFLAGS used to link SWI-Prolog. See [section 10.5](http://www.swi-prolog.org/pldoc/man?section=plld).

**c\_libs**(*atom*, changeable)

Libraries needed to link executables that embed SWI-Prolog. Typically -lswipl if the SWI-Prolog kernel is a shared (DLL). If the SWI-Prolog kernel is in a static library, this flag also contains the dependencies.

**c\_libplso**(*atom*, changeable)

Libraries needed to link extensions (shared object, DLL) to SWI-Prolog. Typically empty on ELF systems and -lswipl on COFF-based systems. See [section 10.5](http://www.swi-prolog.org/pldoc/man?section=plld).

**char\_conversion**(*bool*, changeable)

Determines whether character conversion takes place while reading terms. See also[char\_conversion/2](http://www.swi-prolog.org/pldoc/man?predicate=char_conversion/2).

**character\_escapes**(*bool*, changeable)

If true (default), [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) interprets \ escape sequences in quoted atoms and strings. May be changed. This flag is local to the module in which it is changed.

**colon\_sets\_calling\_context**(*bool*, changeable)

Using the construct <*module*>:<*goal*> sets the *calling context* for executing <*goal*>. This flag is defined by ISO/IEC 13211-2 (Prolog modules standard). See [section 6](http://www.swi-prolog.org/pldoc/man?section=modules).

**color\_term**(*bool*, changeable)

This flag is managed by library library(ansi\_term), which is loaded at startup if the two conditions below are both true. Note that this implies that setting this flag to falsefrom the system or personal initialization file (see [section 2.2](http://www.swi-prolog.org/pldoc/man?section=initfile) disables colored output. The predicate [message\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=message_property/2) can be used to control the actual color scheme depending in the message type passed to [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2).

* stream\_property(current\_output, tty(true))
* \+ current\_prolog\_flag(color\_term, false)

**compile\_meta\_arguments**(*atom*, changeable)

Experimental flag that controls compilation of arguments passed to meta-calls marked `0' or `^' (see [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1)). Supported values are:

**false**

(default). Meta-arguments are passed verbatim.

**control**

Compile meta-arguments that contain control structures ((A,B), (A;B), (A->B;C), etc.). If not compiled at compile time, such arguments are compiled to a temporary clause before execution. Using this option enhances performance of processing complex meta-goals that are known at compile time.

**true**

Also compile references to normal user predicates. This harms performance (a little), but enhances the power of poor-mens consistency check used by [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0)and implemented by [list\_undefined/0](http://www.swi-prolog.org/pldoc/man?predicate=list_undefined/0).

**always**

Always create an intermediate clause, even for system predicates. This prepares for replacing the normal head of the generated predicate with a special reference (similar to database references as used by, e.g., [assert/2](http://www.swi-prolog.org/pldoc/man?predicate=assert/2)) that provides direct access to the executable code, thus avoiding runtime lookup of predicates for meta-calling.

**compiled\_at**(*atom*)

Describes when the system has been compiled. Only available if the C compiler used to compile SWI-Prolog provides the \_\_DATE\_\_ and \_\_TIME\_\_ macros.

**console\_menu**(*bool*)

Set to true in **swipl-win.exe** to indicate that the console supports menus. See also[section 4.33.3](http://www.swi-prolog.org/pldoc/man?section=system).

**cpu\_count**(*integer*, changeable)

Number of physical CPUs or cores in the system. The flag is marked read-write both to allow pretending the system has more or less processors. See also[thread\_setconcurrency/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_setconcurrency/2) and the library library(thread). This flag is not available on systems where we do not know how to get the number of CPUs. This flag is not included in a saved state (see [qsave\_program/1](http://www.swi-prolog.org/pldoc/man?predicate=qsave_program/1)).

**dde**(*bool*)

Set to true if this instance of Prolog supports DDE as described in [section 4.41](http://www.swi-prolog.org/pldoc/man?section=DDE).

**debug**(*bool*, changeable)

Switch debugging mode on/off. If debug mode is activated the system traps encountered spy points (see [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1)) and trace points (see [trace/1](http://www.swi-prolog.org/pldoc/man?predicate=trace/1)). In addition, last-call optimisation is disabled and the system is more conservative in destroying choice points to simplify debugging.

Disabling these optimisations can cause the system to run out of memory on programs that behave correctly if debug mode is off.

**debug\_on\_error**(*bool*, changeable)

If true, start the tracer after an error is detected. Otherwise just continue execution. The goal that raised the error will normally fail. See also [fileerrors/2](http://www.swi-prolog.org/pldoc/man?predicate=fileerrors/2) and the Prolog flagreport\_error. May be changed. Default is true, except for the runtime version.

**debugger\_write\_options**(*term*, changeable)

This argument is given as option-list to [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) for printing goals by the debugger. Modified by the `w', `p' and `<N> d' commands of the debugger. Default is[quoted(true), portray(true), max\_depth(10), attributes(portray)].

**debugger\_show\_context**(*bool*, changeable)

If true, show the context module while printing a stack-frame in the tracer. Normally controlled using the `C' option of the tracer.

**dialect**(*atom*)

Fixed to swi. The code below is a reliable and portable way to detect SWI-Prolog.

is\_dialect(swi) :-

catch(current\_prolog\_flag(dialect, swi), \_, fail).

**double\_quotes**(*codes,chars,atom,string*, changeable)

This flag determines how double quoted strings are read by Prolog and is ---like[character\_escapes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:character_escapes) and [back\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:back_quotes)--- maintained for each module. The default isstring, which produces a string as described in [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings). If **--traditional** is given, the default is codes, which produces a list of character codes, integers that represent a Unicode code-point. The value chars produces a list of one-character atoms and the value atom makes double quotes the same as single quotes, creating a atom. See also[section 5](http://www.swi-prolog.org/pldoc/man?section=extensions).

**editor**(*atom*, changeable)

Determines the editor used by [edit/1](http://www.swi-prolog.org/pldoc/man?predicate=edit/1). See [section 4.5](http://www.swi-prolog.org/pldoc/man?section=listing) for details on selecting the editor used.

**emacs\_inferior\_process**(*bool*)

If true, SWI-Prolog is running as an *inferior process* of (GNU/X-)Emacs. SWI-Prolog assumes this is the case if the environment variable EMACS is t and INFERIOR is yes.

**encoding**(*atom*, changeable)

Default encoding used for opening files in text mode. The initial value is deduced from the environment. See [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for details.

**executable**(*atom*)

Pathname of the running executable. Used by [qsave\_program/2](http://www.swi-prolog.org/pldoc/man?predicate=qsave_program/2) as default emulator.

**exit\_status**(*integer*)

Set by [halt/1](http://www.swi-prolog.org/pldoc/man?predicate=halt/1) to its argument, making the exit status available to hooks registered with[at\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=at_halt/1).

**file\_name\_variables**(*bool*, changeable)

If true (default false), expand $*varname* and ~ in arguments of built-in predicates that accept a file name ([open/3](http://www.swi-prolog.org/pldoc/man?predicate=open/3), [exists\_file/1](http://www.swi-prolog.org/pldoc/man?predicate=exists_file/1), [access\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=access_file/2), etc.). The predicate[expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2) can be used to expand environment variables and wildcard patterns. This Prolog flag is intended for backward compatibility with older versions of SWI-Prolog.

**file\_search\_cache\_time**(*number*, changeable)

Time in seconds for which search results from [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3) are cached. Within this time limit, the system will first check that the old search result satisfies the conditions. Default is 10 seconds, which typically avoids most repetitive searches for (library) files during compilation. Setting this value to 0 (zero) disables the cache.

**gc**(*bool*, changeable)

If true (default), the garbage collector is active. If false, neither garbage collection, nor stack shifts will take place, even not on explicit request. May be changed.

**generate\_debug\_info**(*bool*, changeable)

If true (default) generate code that can be debugged using [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0), [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1), etc. Can be set to false using the **-nodebug**. This flag is scoped within a source file. Many of the libraries have :- set\_prolog\_flag(generate\_debug\_info, false) to hide their details from a normal trace.18

**gmp\_version**(*integer*)

If Prolog is linked with GMP, this flag gives the major version of the GMP library used. See also [section 10.4.8](http://www.swi-prolog.org/pldoc/man?section=foreigninclude).

**gui**(*bool*)

Set to true if XPCE is around and can be used for graphics.

**history**(*integer*, changeable)

If *integer> 0*, support Unix **csh(1)**-like history as described in [section 2.7](http://www.swi-prolog.org/pldoc/man?section=history). Otherwise, only support reusing commands through the command line editor. The default is to set this Prolog flag to 0 if a command line editor is provided (see Prolog flag [readline](http://www.swi-prolog.org/pldoc/man?section=flags#flag:readline)) and 15 otherwise.

**home**(*atom*)

SWI-Prolog's notion of the home directory. SWI-Prolog uses its home directory to find its startup file as <*home*>/boot32.prc (32-bit machines) or <*home*>/boot64.prc (64-bit machines) and to find its library as <*home*>/library.

**hwnd**(*integer*)

In **swipl-win.exe**, this refers to the MS-Windows window handle of the console window.

**integer\_rounding\_function**(*down,toward\_zero*)

ISO Prolog flag describing rounding by // and rem arithmetic functions. Value depends on the C compiler used.

**iso**(*bool*, changeable)

Include some weird ISO compatibility that is incompatible with normal SWI-Prolog behaviour. Currently it has the following effect:

* The //2 (float division) *always* returns a float, even if applied to integers that can be divided.
* In the standard order of terms (see [section 4.7.1](http://www.swi-prolog.org/pldoc/man?section=compare)), all floats are before all integers.
* [atom\_length/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_length/2) yields a type error if the first argument is a number.
* clause/[2,3] raises a permission error when accessing static predicates.
* abolish/[1,2] raises a permission error when accessing static predicates.
* Syntax is closer to the ISO standard:
  + Unquoted commas and bars appearing as atoms are not allowed. Instead off(,,a) now write f(',',a). Unquoted commas can only be used to separate arguments in functional notation and list notation, and as a conjunction operator. Unquoted bars can only appear within lists to separate head and tail, like [Head|Tail], and as infix operator for alternation in grammar rules, like a --> b | c.
  + Within functional notation and list notation terms must have priority below 1000. That means that rules and control constructs appearing as arguments need bracketing. A term like [a :- b, c]. must now be disambiguated to mean [(a :- b), c]. or [(a :- b, c)].
  + Operators appearing as operands must be bracketed. Instead of X == -, true. write X == (-), true. Currently, this is not entirely enforced.
  + Backslash-escaped newlines are interpreted according to the ISO standard. See [section 2.15.2.1](http://www.swi-prolog.org/pldoc/man?section=syntax).

**large\_files**(*bool*)

If present and true, SWI-Prolog has been compiled with *large file support* (LFS) and is capable of accessing files larger than 2GB on 32-bit hardware. Large file support is default on installations built using **configure** that support it and may be switched off using the configure option --disable-largefile.

**last\_call\_optimisation**(*bool*, changeable)

Determines whether or not last-call optimisation is enabled. Normally the value of this flag is the negation of the [debug](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debug) flag. As programs may run out of stack if last-call optimisation is omitted, it is sometimes necessary to enable it during debugging.

**max\_arity**(*unbounded*)

ISO Prolog flag describing there is no maximum arity to compound terms.

**max\_integer**(*integer*)

Maximum integer value if integers are *bounded*. See also the flag [bounded](http://www.swi-prolog.org/pldoc/man?section=flags#flag:bounded) and [section 4.26.2.1](http://www.swi-prolog.org/pldoc/man?section=arith).

**max\_tagged\_integer**(*integer*)

Maximum integer value represented as a `tagged' value. Tagged integers require one word storage. Larger integers are represented as `indirect data' and require significantly more space.

**min\_integer**(*integer*)

Minimum integer value if integers are *bounded*. See also the flag [bounded](http://www.swi-prolog.org/pldoc/man?section=flags#flag:bounded) and [section 4.26.2.1](http://www.swi-prolog.org/pldoc/man?section=arith).

**min\_tagged\_integer**(*integer*)

Start of the tagged-integer value range.

**occurs\_check**(*atom*, changeable)

This flag controls unification that creates an infinite tree (also called *cyclic term*) and can have three values. Using false (default), unification succeeds, creating an infinite tree. Using true, unification behaves as [unify\_with\_occurs\_check/2](http://www.swi-prolog.org/pldoc/man?predicate=unify_with_occurs_check/2), failing silently. Usingerror, an attempt to create a cyclic term results in an occurs\_check exception. The latter is intended for debugging unintentional creations of cyclic terms. Note that this flag is a global flag modifying fundamental behaviour of Prolog. Changing the flag from its default may cause libraries to stop functioning properly.

**open\_shared\_object**(*bool*)

If true, [open\_shared\_object/2](http://www.swi-prolog.org/pldoc/man?predicate=open_shared_object/2) and friends are implemented, providing access to shared libraries (.so files) or dynamic link libraries (.DLL files).

**optimise**(*bool*, changeable)

If true, compile in optimised mode. The initial value is true if Prolog was started with the **-O** command line option. The [optimise](http://www.swi-prolog.org/pldoc/man?section=flags#flag:optimise) flag is scoped to a source file.

Currently optimised compilation implies compilation of arithmetic, and deletion of redundant [true/0](http://www.swi-prolog.org/pldoc/man?predicate=true/0) that may result from [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2).

Later versions might imply various other optimisations such as integrating small predicates into their callers, eliminating constant expressions and other predictable constructs. Source code optimisation is never applied to predicates that are declared dynamic (see [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1)).

**os\_argv**(*list*, changeable)

List is a list of atoms representing the command line arguments used to invoke SWI-Prolog. Please note that **all** arguments are included in the list returned. See [argv](http://www.swi-prolog.org/pldoc/man?section=flags#flag:argv) to get the application options.

**pid**(*int*)

Process identifier of the running Prolog process. Existence of this flag is implementation-defined.

**pipe**(*bool*, changeable)

If true, open(pipe(command), mode, Stream), etc. are supported. Can be changed to disable the use of pipes in applications testing this feature. Not recommended.

**print\_write\_options**(*term*, changeable)

Specifies the options for [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) used by [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) and [print/2](http://www.swi-prolog.org/pldoc/man?predicate=print/2).

**prompt\_alternatives\_on**(*atom*, changeable)

Determines prompting for alternatives in the Prolog top level. Default is determinism, which implies the system prompts for alternatives if the goal succeeded while leaving choice points. Many classical Prolog systems behave as groundness: they prompt for alternatives if and only if the query contains variables.

**qcompile**(*atom*, changeable)

This option provides the default for the qcompile(+Atom) option of [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2).

**readline**(*bool*)

If true, SWI-Prolog is linked with the readline library. This is done by default if you have this library installed on your system. It is also true for the Win32 swipl-win.exe version of SWI-Prolog, which realises a subset of the readline functionality.

**resource\_database**(*atom*)

Set to the absolute filename of the attached state. Typically this is the file boot32.prc, the file specified with **-x** or the running executable. See also [resource/3](http://www.swi-prolog.org/pldoc/man?predicate=resource/3).

**report\_error**(*bool*, changeable)

If true, print error messages; otherwise suppress them. May be changed. See also the[debug\_on\_error](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debug_on_error) Prolog flag. Default is true, except for the runtime version.

**runtime**(*bool*)

If present and true, SWI-Prolog is compiled with -DO\_RUNTIME, disabling various useful development features (currently the tracer and profiler).

**sandboxed\_load**(*bool*, changeable)

If true (default false), [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) calls hooks to allow library(sandbox) to verify the safety of directives.

**saved\_program**(*bool*)

If present and true, Prolog has been started from a state saved withqsave\_program/[1,2].

**shared\_object\_extension**(*atom*)

Extension used by the operating system for shared objects. .so for most Unix systems and .dll for Windows. Used for locating files using the file\_type executable. See also[absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3).

**shared\_object\_search\_path**(*atom*)

Name of the environment variable used by the system to search for shared objects.

**signals**(*bool*)

Determine whether Prolog is handling signals (software interrupts). This flag is false if the hosting OS does not support signal handling or the command line option **-nosignals** is active. See [section 10.4.21.1](http://www.swi-prolog.org/pldoc/man?section=foreigninclude) for details.

**stream\_type\_check**(*atom*, changeable)

Defines whether and how strictly the system validates that byte I/O should not be applied to text streams and text I/O should not be applied to binary streams. Values arefalse (no checking), true (full checking) and loose. Using checking mode loose(default), the system accepts byte I/O from text stream that use ISO Latin-1 encoding and accepts writing text to binary streams.

**system\_thread\_id**(*int*)

Available in multithreaded version (see [section 9](http://www.swi-prolog.org/pldoc/man?section=threads)) where the operating system provides system-wide integer thread identifiers. The integer is the thread identifier used by the operating system for the calling thread. See also [thread\_self/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_self/1).

**timezone**(*integer*)

Offset in seconds west of GMT of the current time zone. Set at initialization time from the timezone variable associated with the POSIX tzset() function. See also[convert\_time/2](http://www.swi-prolog.org/pldoc/man?predicate=convert_time/2).

**toplevel\_print\_anon**(*bool*, changeable)

If true, top-level variables starting with an underscore (\_) are printed normally. If falsethey are hidden. This may be used to hide bindings in complex queries from the top level.

**toplevel\_print\_factorized**(*bool*, changeable)

If true (default false) show the internal sharing of subterms in the answer substitution. The example below reveals internal sharing of leaf nodes in *red-black trees* as implemented by the library(rbtrees) predicate rb\_new/1:

?- set\_prolog\_flag(toplevel\_print\_factorized, true).

?- rb\_new(X).

X = t(\_S1, \_S1), % where

\_S1 = black('', \_G387, \_G388, '').

If this flag is false, the % where notation is still used to indicate cycles as illustrated below. This example also shows that the implementation reveals the internal cycle length, and *not* the minimal cycle length. Cycles of different length are indistinguishable in Prolog (as illustrated by S == R).

?- S = s(S), R = s(s(R)), S == R.

S = s(S),

R = s(s(R)).

**answer\_write\_options**(*term*, changeable)

This argument is given as option-list to [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) for printing results of queries. Default is [quoted(true), portray(true), max\_depth(10), attributes(portray)].

**toplevel\_prompt**(*atom*, changeable)

Define the prompt that is used by the interactive top level. The following ~ (tilde) sequences are replaced:

|  |  |
| --- | --- |
| ~m | *Type in* module if not user (see [module/1](http://www.swi-prolog.org/pldoc/man?predicate=module/1)) |
| ~l | *Break level* if not 0 (see [break/0](http://www.swi-prolog.org/pldoc/man?predicate=break/0)) |
| ~d | *Debugging state* if not normal execution (see [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0), [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0)) |
| ~! | *History event* if history is enabled (see flag [history](http://www.swi-prolog.org/pldoc/man?section=flags#flag:history)) |

**toplevel\_var\_size**(*int*, changeable)

Maximum size counted in literals of a term returned as a binding for a variable in a top-level query that is saved for re-use using the $ variable reference. See [section 2.8](http://www.swi-prolog.org/pldoc/man?section=topvars).

**trace\_gc**(*bool*, changeable)

If true (default false), garbage collections and stack-shifts will be reported on the terminal. May be changed. Values are reported in bytes as *G*+*T*, where *G* is the global stack value and *T* the trail stack value. `Gained' describes the number of bytes reclaimed. `used' the number of bytes on the stack after GC and `free' the number of bytes allocated, but not in use. Below is an example output.

% GC: gained 236,416+163,424 in 0.00 sec;

used 13,448+5,808; free 72,568+47,440

**traditional**(*bool*)

Available in SWI-Prolog version 7. If true, `traditional' mode has been selected using **--traditional**. See also [section 5](http://www.swi-prolog.org/pldoc/man?section=extensions).

**tty\_control**(*bool*, changeable)

Determines whether the terminal is switched to raw mode for [get\_single\_char/1](http://www.swi-prolog.org/pldoc/man?predicate=get_single_char/1), which also reads the user actions for the trace. May be set. See also the **+/-tty** command line option.

**unix**(*bool*)

If present and true, the operating system is some version of Unix. Defined if the C compiler used to compile this version of SWI-Prolog either defines \_\_unix\_\_ or unix. On other systems this flag is not available. See also [apple](http://www.swi-prolog.org/pldoc/man?section=flags#flag:apple) and [windows](http://www.swi-prolog.org/pldoc/man?section=flags#flag:windows).

**unknown**(*fail,warning,error*, changeable)

Determines the behaviour if an undefined procedure is encountered. If fail, the predicate fails silently. If warn, a warning is printed, and execution continues as if the predicate was not defined, and if error (default), an existence\_error exception is raised. This flag is local to each module and inherited from the module's *import-module*. Using default setup, this implies that normal modules inherit the flag fromuser, which in turn inherit the value error from system. The user may change the flag for module user to change the default for all application modules or for a specific module. It is strongly advised to keep the error default and use [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1) and/or[multifile/1](http://www.swi-prolog.org/pldoc/man?predicate=multifile/1) to specify possible non-existence of a predicate.

**unload\_foreign\_libraries**(*bool*, changeable)

If true (default false), unload all loaded foreign libraries. Default is false because modern OSes reclaim the resources anyway and unloading the foreign code may cause registered hooks to point to no longer existing data or code.

**user\_flags**(*Atom*, changeable)

Define the behaviour of [set\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=set_prolog_flag/2) if the flag is not known. Values are silent,warning and error. The first two create the flag on-the-fly, where warning prints a message. The value error is consistent with ISO: it raises an existence error and does not create the flag. See also [create\_prolog\_flag/3](http://www.swi-prolog.org/pldoc/man?predicate=create_prolog_flag/3). The default is silent, but future versions may change that. Developers are encouraged to use another value and ensure proper use of [create\_prolog\_flag/3](http://www.swi-prolog.org/pldoc/man?predicate=create_prolog_flag/3) to create flags for their library.

**verbose**(*Atom*, changeable)

This flag is used by [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). If its value is silent, messages of typeinformational and banner are suppressed. The **-q** switches the value from the initialnormal to silent.

**verbose\_autoload**(*bool*, changeable)

If true the normal consult message will be printed if a library is autoloaded. By default this message is suppressed. Intended to be used for debugging purposes.

**verbose\_load**(*atom*, changeable)

Determines messages printed for loading (compiling) Prolog files. Current values arefull (print a message at the start and end of each file loaded), normal (print a message at the end of each file loaded), brief (print a message at end of loading the toplevel file), and silent (no messages are printed, default). The value of this flag is normally controlled by the option silent(Bool) provided by [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2).

**verbose\_file\_search**(*bool*, changeable)

If true (default false), print messages indicating the progress ofabsolute\_file\_name/[2,3] in locating files. Intended for debugging complicated file-search paths. See also [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2).

**version**(*integer*)

The version identifier is an integer with value:

*10000 × Major + 100 × Minor + Patch*

Note that in releases up to 2.7.10 this Prolog flag yielded an atom holding the three numbers separated by dots. The current representation is much easier for implementing version-conditional statements.

**version\_data**(*swi(Major, Minor, Patch, Extra)*)

Part of the dialect compatibility layer; see also the Prolog flag [dialect](http://www.swi-prolog.org/pldoc/man?section=flags#flag:dialect) and [section C](http://www.swi-prolog.org/pldoc/man?section=dialect).*Extra* provides platform-specific version information. Currently it is simply unified to [].

**version\_git**(*atom*)

Available if created from a git repository. See **git-describe** for details.

**warn\_override\_implicit\_import**(*bool*, changeable)

If true (default), a warning is printed if an implicitly imported predicate is clobbered by a local definition. See [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) for details.

**win\_file\_access\_check**(*atom*, changeable)

Controls the behaviour or [access\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=access_file/2) under Windows. There is no reliable way to check access to files and directories on Windows. This flag allows for switching between three alternative approximations.

**access**

Use Windows \_waccess() function. This ignores ACLs (Access Control List) and thus may indicate that access is allowed while it is not.

**filesecurity**

Use the Windows GetFileSecurity() function. This does not work on all file systems, but is probably the best choice on file systems that do support it, notably local NTFS volumes.

**openclose**

Try to open the file and close it. This works reliable for files, but not for directories. Currently directories are checked using \_waccess(). This is the default.

**windows**(*bool*)

If present and true, the operating system is an implementation of Microsoft Windows (NT/2000/XP, etc.). This flag is only available on MS-Windows based versions.

**write\_attributes**(*atom*, changeable)

Defines how [write/1](http://www.swi-prolog.org/pldoc/man?predicate=write/1) and friends write attributed variables. The option values are described with the attributes option of [write\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=write_term/3). Default is ignore.

**write\_help\_with\_overstrike**(*bool*)

Internal flag used by [help/1](http://www.swi-prolog.org/pldoc/man?predicate=help/1) when writing to a terminal. If present and true it prints bold and underlined text using *overstrike*.

**xpce**(*bool*)

Available and set to true if the XPCE graphics system is loaded.

**xpce\_version**(*atom*)

Available and set to the version of the loaded XPCE system.

**file\_base\_name**(*+File, -BaseName*)

Extracts the filename part from a path specification. If *File* does not contain any directory separators, *File* is returned in *BaseName*. See also [file\_directory\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=file_directory_name/2). If the *File*arguments ends with a /, e.g., '/hello/', *BaseName* is unified with the empty atom ('').

**file\_directory\_name**(*+File, -Directory*)

Extracts the directory part of *File*. The returned *Directory* name does not end in /. There are two special cases. The directory name of / is / itself, and the directory name is . if *File* does not contain any / characters. If the *File* argument ends with a /, e.g., '/hello/', it is not a valid file name. In this case the final / is removed from *File*, e.g., '/hello'.

See also [directory\_file\_path/3](http://www.swi-prolog.org/pldoc/man?predicate=directory_file_path/3) from library(filesex). The system ensures that for every valid *Path* using the Prolog (POSIX) directory separators, following is true on systems with a sound implementation of [same\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=same_file/2).121 See also [prolog\_to\_os\_filename/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_to_os_filename/2).

...,

file\_directory\_name(Path, Dir),

file\_base\_name(Path, File),

directory\_file\_path(Dir, File, Path2),

same\_file(Path, Path2).

**file\_name\_extension**(*?Base, ?Extension, ?Name*)

This predicate is used to add, remove or test filename extensions. The main reason for its introduction is to deal with different filename properties in a portable manner. If the file system is case-insensitive, testing for an extension will also be done case-insensitive.*Extension* may be specified with or without a leading dot (.). If an *Extension* is generated, it will not have a leading dot.

**file\_search\_path**(*+Alias, ?Path*)

Dynamic predicate used to specify `path aliases'. This feature is best described using an example. Given the definition:

file\_search\_path(demo, '/usr/lib/prolog/demo').

the file specification demo(myfile) will be expanded to /usr/lib/prolog/demo/myfile. The second argument of [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2) may be another alias.

Below is the initial definition of the file search path. This path implies swi(<*Path*>) and refers to a file in the SWI-Prolog home directory. The alias foreign(<*Path*>) is intended for storing shared libraries (.so or .DLL files). See also [use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1).

user:file\_search\_path(library, X) :-

library\_directory(X).

user:file\_search\_path(swi, Home) :-

current\_prolog\_flag(home, Home).

user:file\_search\_path(foreign, swi(ArchLib)) :-

current\_prolog\_flag(arch, Arch),

atom\_concat('lib/', Arch, ArchLib).

user:file\_search\_path(foreign, swi(lib)).

user:file\_search\_path(path, Dir) :-

getenv('PATH', Path),

( current\_prolog\_flag(windows, true)

-> atomic\_list\_concat(Dirs, (;), Path)

; atomic\_list\_concat(Dirs, :, Path)

),

member(Dir, Dirs).

The [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2) expansion is used by all loading predicates as well as byabsolute\_file\_name/[2,3].

The Prolog flag [verbose\_file\_search](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose_file_search) can be set to true to help debugging Prolog's search for files.

**find\_chr\_constraint**(*-Constraint*)

Returns a constraint in the constraint store. Via backtracking, all constraints in the store can be enumerated.

**findall**(*+Template, :Goal, -Bag*)

Create a list of the instantiations *Template* gets successively on backtracking over *Goal* and unify the result with *Bag*. Succeeds with an empty list if *Goal* has no solutions. [findall/3](http://www.swi-prolog.org/pldoc/man?predicate=findall/3) is equivalent to [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3) with all free variables bound with the existential operator (**^**), except that [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3) fails when *Goal* has no solutions.

**findall**(*+Template, :Goal, -Bag, +Tail*)

As [findall/3](http://www.swi-prolog.org/pldoc/man?predicate=findall/3), but returns the result as the difference list *Bag*-*Tail*. The 3-argument version is defined as

findall(Templ, Goal, Bag) :-

findall(Templ, Goal, Bag, [])

**findnsols**(*+N, @Template, :Goal, -List*)

*[nondet]***findnsols**(*+N, @Template, :Goal, -List, ?Tail*)

As [findall/3](http://www.swi-prolog.org/pldoc/man?predicate=findall/3) and [findall/4](http://www.swi-prolog.org/pldoc/man?predicate=findall/4), but generates at most *N* solutions. If *N* solutions are returned, this predicate succeeds with a choice point if *Goal* has a choice point. Backtracking returns the next chunk of (at most) *N* solutions. In addition to passing a plain integer for *N*, a term of the form count(N) is accepted. Using count(N), the size of the next chunk can be controlled using[nb\_setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_setarg/3). The non-deterministic behaviour used to implement the *chunk* option inlibrary(pengines). Based on Ciao, but the Ciao version is deterministic. Portability can be achieved by wrapping the goal in [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1). Below are three examples. The first illustrates standard chunking of answers. The second illustrates that the chunk size can be adjusted dynamically and the last illustrates that no choice point is left if *Goal* leaves no choice-point after the last solution.

?- findnsols(5, I, between(1, 12, I), L).

L = [1, 2, 3, 4, 5] ;

L = [6, 7, 8, 9, 10] ;

L = [11, 12].

?- State = count(2),

findnsols(State, I, between(1, 12, I), L),

nb\_setarg(1, State, 5).

State = count(5), L = [1, 2] ;

State = count(5), L = [3, 4, 5, 6, 7] ;

State = count(5), L = [8, 9, 10, 11, 12].

?- findnsols(4, I, between(1, 4, I), L).

L = [1, 2, 3, 4].

# [findnsols/5](http://www.swi-prolog.org/pldoc/allsolutions.html#findnsols/5)

**flag**(*+Key, -Old, +New*)

True when *Old* is the current value of the flag *Key* and the flag has been set to *New*. *New* can be an arithmetic expression. The update is *atomic*. This predicate can be used to create a *shared*global counter as illustrated in the example below.

next\_id(Id) :-

flag(my\_id, Id, Id+1).

**float**(*@Term*)

True if *Term* is bound to a floating point number.

**flush\_output**

Flush pending output on current output stream. [flush\_output/0](http://www.swi-prolog.org/pldoc/man?predicate=flush_output/0) is automatically generated by[read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) and derivatives if the current input stream is user and the cursor is not at the left margin.

**flush\_output**(*+Stream*)

Flush output on the specified stream. The stream must be open for writing.

**forall**(*:Cond, :Action*)

For all alternative bindings of *Cond*, *Action* can be proven. The example verifies that all arithmetic statements in the given list are correct. It does not say which is wrong if one proves wrong.

?- forall(member(Result = Formula, [2 = 1 + 1, 4 = 2 \* 2]),

Result =:= Formula).

The predicate [forall/2](http://www.swi-prolog.org/pldoc/man?predicate=forall/2) is implemented as \+ ( Cond, \+ Action), i.e., *There is no instantiation of Cond for which Action is false.*. The use of double negation implies that[forall/2](http://www.swi-prolog.org/pldoc/man?predicate=forall/2) *does not change any variable bindings*. It proves a relation. The [forall/2](http://www.swi-prolog.org/pldoc/man?predicate=forall/2) control structure can be used for its side-effects. E.g., the following asserts relations in a list into the dynamic database:

?- forall(member(Child-Parent, ChildPairs),

assertz(child\_of(Child, Parent))).

Using [forall/2](http://www.swi-prolog.org/pldoc/man?predicate=forall/2) as forall(Generator, SideEffect) is preferred over the classical *failure driven loop* as shown below because it makes it explicit which part of the construct is the generator and which part creates the side effects. Also, unexpected failure of the side effect causes the construct to fail. Failure makes it evident that there is an issue with the code, while a failure driven loop would succeed with an erroneous result.

...,

( Generator,

SideEffect,

fail

; true

)

If your intent is to create variable bindings, the [forall/2](http://www.swi-prolog.org/pldoc/man?predicate=forall/2) control structure is inadequate. Possibly you are looking for [maplist/2](http://www.swi-prolog.org/pldoc/man?predicate=maplist/2), [findall/3](http://www.swi-prolog.org/pldoc/man?predicate=findall/3) or [foreach/2](http://www.swi-prolog.org/pldoc/man?predicate=foreach/2).

**format**(*+Format*)

Defined as `format(Format) :- format(Format, []).'. See [format/2](http://www.swi-prolog.org/pldoc/man?predicate=format/2) for details.

**format**(*+Format, :Arguments*)

*Format* is an atom, list of character codes, or a Prolog string. *Arguments* provides the arguments required by the format specification. If only one argument is required and this single argument is not a list, the argument need not be put in a list. Otherwise the arguments are put in a list.

Special sequences start with the tilde (~), followed by an optional numeric argument, optionally followed by a colon modifier (:), 115 followed by a character describing the action to be undertaken. A numeric argument is either a sequence of digits, representing a positive decimal number, a sequence `<*character*>, representing the character code value of the character (only useful for ~t) or a asterisk (\*), in which case the numeric argument is taken from the next argument of the argument list, which should be a positive integer. E.g., the following three examples all pass 46 (.) to ~t:

?- format('~w ~46t ~w~72|~n', ['Title', 'Page']).

?- format('~w ~`.t ~w~72|~n', ['Title', 'Page']).

?- format('~w ~\*t ~w~72|~n', ['Title', 46, 'Page']).

Numeric conversion (d, D, e, E, f, g and G) accept an arithmetic expression as argument. This is introduced to handle rational numbers transparently (see [section 4.26.2.2](http://www.swi-prolog.org/pldoc/man?section=arith)). The floating point conversions allow for unlimited precision for printing rational numbers in decimal form. E.g., the following will write as many 3's as you want by changing the `50'.

?- format('~50f', [10 rdiv 3]).

3.33333333333333333333333333333333333333333333333333

* ~
* Output the tilde itself.
* a
* Output the next argument, which must be an atom. This option is equivalent to **w**, except that it requires the argument to be an atom.
* c
* Interpret the next argument as a character code and add it to the output. This argument must be a valid Unicode character code. Note that the actually emitted bytes are defined by the character encoding of the output stream and an exception may be raised if the output stream is not capable of representing the requested Unicode character. See[section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for details.
* d
* Output next argument as a decimal number. It should be an integer. If a numeric argument is specified, a dot is inserted *argument* positions from the right (useful for doing fixed point arithmetic with integers, such as handling amounts of money).

The colon modifier (e.g., ~:d) causes the number to be printed according to the locale of the output stream. See [section 4.22](http://www.swi-prolog.org/pldoc/man?section=locale).

* D
* Same as **d**, but makes large values easier to read by inserting a comma every three digits left or right of the dot. This is the same as ~:d, but using the fixed English locale.
* e
* Output next argument as a floating point number in exponential notation. The numeric argument specifies the precision. Default is 6 digits. Exact representation depends on the C library function printf(). This function is invoked with the format %.<*precision*>e.
* E
* Equivalent to **e**, but outputs a capital E to indicate the exponent.
* f
* Floating point in non-exponential notation. The numeric argument defines the number of digits right of the decimal point. If the colon modifier (:) is used, the float is formatted using conventions from the current locale, which may define the decimal point as well as grouping of digits left of the decimal point.
* g
* Floating point in **e** or **f** notation, whichever is shorter.
* G
* Floating point in **E** or **f** notation, whichever is shorter.
* i
* Ignore next argument of the argument list. Produces no output.
* I
* Emit a decimal number using Prolog digit grouping (the underscore, \_). The argument describes the size of each digit group. The default is 3. See also [section 2.15.2.3](http://www.swi-prolog.org/pldoc/man?section=syntax). For example:
* ?- A is 1<<100, format('~10I', [A]).

1\_2676506002\_2822940149\_6703205376

* k
* Give the next argument to [write\_canonical/1](http://www.swi-prolog.org/pldoc/man?predicate=write_canonical/1).
* n
* Output a newline character.
* N
* Only output a newline if the last character output on this stream was not a newline. Not properly implemented yet.
* p
* Give the next argument to [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1).
* q
* Give the next argument to [writeq/1](http://www.swi-prolog.org/pldoc/man?predicate=writeq/1).
* r
* Print integer in radix numeric argument notation. Thus ~16r prints its argument hexadecimal. The argument should be in the range *[2, ... , 36]*. Lowercase letters are used for digits above 9. The colon modifier may be used to form locale-specific digit groups.
* R
* Same as **r**, but uses uppercase letters for digits above 9.
* s
* Output text from a list of character codes or a string (see [string/1](http://www.swi-prolog.org/pldoc/man?predicate=string/1) and [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)) from the next argument.116
* @
* Interpret the next argument as a goal and execute it. Output written to thecurrent\_output stream is inserted at this place. Goal is called in the module calling[format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3). This option is not present in the original definition by Quintus, but supported by some other Prolog systems.
* t
* All remaining space between 2 tab stops is distributed equally over ~t statements between the tab stops. This space is padded with spaces by default. If an argument is supplied, it is taken to be the character code of the character used for padding. This can be used to do left or right alignment, centering, distributing, etc. See also ~| and ~+ to set tab stops. A tab stop is assumed at the start of each line.
* |
* Set a tab stop on the current position. If an argument is supplied set a tab stop on the position of that argument. This will cause all ~t's to be distributed between the previous and this tab stop.
* +
* Set a tab stop (as ~|) relative to the last tab stop or the beginning of the line if no tab stops are set before the ~+. This constructs can be used to fill fields. The partial format sequence below prints an integer right-aligned and padded with zeros in 6 columns. The ... sequences in the example illustrate that the integer is aligned in 6 columns regardless of the remainder of the format specification.

format('...~|~`0t~d~6+...', [..., Integer, ...])

* w
* Give the next argument to [write/1](http://www.swi-prolog.org/pldoc/man?predicate=write/1).
* W
* Give the next two arguments to [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2). For example, format('~W', [Term, [numbervars(true)]]). This option is SWI-Prolog specific.

Example:

simple\_statistics :-

<obtain statistics> % left to the user

format('~tStatistics~t~72|~n~n'),

format('Runtime: ~`.t ~2f~34| Inferences: ~`.t ~D~72|~n',

[RunT, Inf]),

....

will output

Statistics

Runtime: .................. 3.45 Inferences: .......... 60,345

**format**(*+Output, +Format, :Arguments*)

As [format/2](http://www.swi-prolog.org/pldoc/man?predicate=format/2), but write the output on the given *Output*. The de-facto standard only allows*Output* to be a stream. The SWI-Prolog implementation allows all valid arguments for[with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2).117 For example:

?- format(atom(A), '~D', [1000000]).

A = '1,000,000'

**format\_time**(*+Out, +Format, +StampOrDateTime*)

Modelled after POSIX strftime(), using GNU extensions. *Out* is a destination as specified with[with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2). *Format* is an atom or string with the following conversions. Conversions start with a tilde (%) character.120 *StampOrDateTime* is either a numeric time-stamp, a termdate(Y,M,D,H,M,S,O,TZ,DST) or a term date(Y,M,D).

* a
* The abbreviated weekday name according to the current locale. Use [format\_time/4](http://www.swi-prolog.org/pldoc/man?predicate=format_time/4) for POSIX locale.
* A
* The full weekday name according to the current locale. Use [format\_time/4](http://www.swi-prolog.org/pldoc/man?predicate=format_time/4) for POSIX locale.
* b
* The abbreviated month name according to the current locale. Use [format\_time/4](http://www.swi-prolog.org/pldoc/man?predicate=format_time/4) for POSIX locale.
* B
* The full month name according to the current locale. Use [format\_time/4](http://www.swi-prolog.org/pldoc/man?predicate=format_time/4) for POSIX locale.
* c
* The preferred date and time representation for the current locale.
* C
* The century number (year/100) as a 2-digit integer.
* d
* The day of the month as a decimal number (range 01 to 31).
* D
* Equivalent to %m/%d/%y. (For Americans only. Americans should note that in other countries %d/%m/%y is rather common. This means that in an international context this format is ambiguous and should not be used.)
* e
* Like %d, the day of the month as a decimal number, but a leading zero is replaced by a space.
* E
* Modifier. Not implemented.
* f
* Number of microseconds. The f can be prefixed by an integer to print the desired number of digits. E.g., %3f prints milliseconds. This format is not covered by any standard, but available with different format specifiers in various incarnations of the strftime() function.
* F
* Equivalent to %Y-%m-%d (the ISO 8601 date format).
* g
* Like %G, but without century, i.e., with a 2-digit year (00-99).
* G
* The ISO 8601 year with century as a decimal number. The 4-digit year corresponding to the ISO week number (see %V). This has the same format and value as %y, except that if the ISO week number belongs to the previous or next year, that year is used instead.
* V
* The ISO 8601:1988 week number of the current year as a decimal number, range 01 to 53, where week 1 is the first week that has at least 4 days in the current year, and with Monday as the first day of the week. See also %U and %W.
* h
* Equivalent to %b.
* H
* The hour as a decimal number using a 24-hour clock (range 00 to 23).
* I
* The hour as a decimal number using a 12-hour clock (range 01 to 12).
* j
* The day of the year as a decimal number (range 001 to 366).
* k
* The hour (24-hour clock) as a decimal number (range 0 to 23); single digits are preceded by a blank. (See also %H.)
* l
* The hour (12-hour clock) as a decimal number (range 1 to 12); single digits are preceded by a blank. (See also %I.)
* m
* The month as a decimal number (range 01 to 12).
* M
* The minute as a decimal number (range 00 to 59).
* n
* A newline character.
* O
* Modifier to select locale-specific output. Not implemented.
* p
* Either `AM' or `PM' according to the given time value, or the corresponding strings for the current locale. Noon is treated as `pm' and midnight as `am'.
* P
* Like %p but in lowercase: `am' or `pm' or a corresponding string for the current locale.
* r
* The time in a.m. or p.m. notation. In the POSIX locale this is equivalent to `%I:%M:%S %p'.
* R
* The time in 24-hour notation (%H:%M). For a version including the seconds, see %T below.
* s
* The number of seconds since the Epoch, i.e., since 1970-01-01 00:00:00 UTC.
* S
* The second as a decimal number (range 00 to 60). (The range is up to 60 to allow for occasional leap seconds.)
* t
* A tab character.
* T
* The time in 24-hour notation (%H:%M:%S).
* u
* The day of the week as a decimal, range 1 to 7, Monday being 1. See also %w.
* U
* The week number of the current year as a decimal number, range 00 to 53, starting with the first Sunday as the first day of week 01. See also %V and %W.
* w
* The day of the week as a decimal, range 0 to 6, Sunday being 0. See also %u.
* W
* The week number of the current year as a decimal number, range 00 to 53, starting with the first Monday as the first day of week 01.
* x
* The preferred date representation for the current locale without the time.
* X
* The preferred time representation for the current locale without the date.
* y
* The year as a decimal number without a century (range 00 to 99).
* Y
* The year as a decimal number including the century.
* z
* The timezone as hour offset from GMT using the format HHmm. Required to emit RFC822-conforming dates (using '%a, %d %b %Y %T %z'). Our implementation supports %:z, which modifies the output to HH:mm as required by XML-Schema. Note that both notations are valid in ISO 8601. The sequence %:z is compatible to the GNU date(1) command.
* Z
* The timezone or name or abbreviation.
* +
* The date and time in date(1) format.
* %
* A literal `%' character.

The table below gives some format strings for popular time representations. RFC1123 is used by HTTP. The full implementation of [http\_timestamp/2](http://www.swi-prolog.org/pldoc/man?predicate=http_timestamp/2) as available fromlibrary(http/http\_header) is here.

http\_timestamp(Time, Atom) :-

stamp\_date\_time(Time, Date, 'UTC'),

format\_time(atom(Atom),

'%a, %d %b %Y %T GMT',

Date, posix).

|  |  |
| --- | --- |
| **Standard** | **Format string** |
| **xsd** | '%FT%T%:z' |
| **ISO8601** | '%FT%T%z' |
| **RFC822** | '%a, %d %b %Y %T %z' |
| **RFC1123** | '%a, %d %b %Y %T GMT' |

**format\_time**(*+Out, +Format, +StampOrDateTime, +Locale*)

Format time given a specified *Locale*. This predicate is a work-around for lacking proper portable and thread-safe time and locale handling in current C libraries. In its current implementation the only value allowed for *Locale* is posix, which currently only modifies the behaviour of the a, A, b and B format specifiers. The predicate is used to be able to emit POSIX locale week and month names for emitting standardised time-stamps such as RFC1123.

**format\_predicate**(*+Char, +Head*)

If a sequence ~c (tilde, followed by some character) is found, the [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3) and friends first check whether the user has defined a predicate to handle the format. If not, the built-in formatting rules described above are used. *Char* is either a character code or a one-character atom, specifying the letter to be (re)defined. *Head* is a term, whose name and arity are used to determine the predicate to call for the redefined formatting character. The first argument to the predicate is the numeric argument of the format command, or the atom default if no argument is specified. The remaining arguments are filled from the argument list. The example below defines ~T to print a timestamp in ISO8601 format (see [format\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=format_time/3)). The subsequent block illustrates a possible call.

:- format\_predicate('T', format\_time(\_Arg,\_Time)).

format\_time(\_Arg, Stamp) :-

must\_be(number, Stamp),

format\_time(current\_output, '%FT%T%z', Stamp).

?- get\_time(Now),

format('Now, it is ~T~n', [Now]).

Now, it is 2012-06-04T19:02:01+0200

Now = 1338829321.6620328.

**term\_attvars**(*+Term, -AttVars*)

*AttVars* is a list of all attributed variables in *Term* and its attributes. That is, [term\_attvars/2](http://www.swi-prolog.org/pldoc/man?predicate=term_attvars/2)works recursively through attributes. This predicate is cycle-safe. The goalterm\_attvars(Term,[]) in an efficient test that *Term* has *no* attributes; scanning the term is aborted after the first attributed variable is found.

**term\_variables**(*+Term, -List*)

Unify *List* with a list of variables, each sharing with a unique variable of *Term*.88 The variables in *List* are ordered in order of appearance traversing *Term* depth-first and left-to-right. See also [term\_variables/3](http://www.swi-prolog.org/pldoc/man?predicate=term_variables/3). For example:

?- term\_variables(a(X, b(Y, X), Z), L).

L = [X, Y, Z].

**term\_variables**(*+Term, -List, ?Tail*)

Difference list version of [term\_variables/2](http://www.swi-prolog.org/pldoc/man?predicate=term_variables/2). That is, *Tail* is the tail of the variable list *List*.

**text\_to\_string**(*+Text, -String*)

Converts *Text* to a string. *Text* is an atom, string or list of characters (codes or chars). When running in **--traditional** mode, '[]' is ambiguous and interpreted as an empty string.

**freeze**(*+Var, :Goal*)

Delay the execution of *Goal* until *Var* is bound (i.e. is not a variable or attributed variable). If*Var* is bound on entry [freeze/2](http://www.swi-prolog.org/pldoc/man?predicate=freeze/2) is equivalent to [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1). The [freeze/2](http://www.swi-prolog.org/pldoc/man?predicate=freeze/2) predicate is realised using an attributed variable associated with the module freeze. Use frozen(Var, Goal) to find out whether and which goals are delayed on *Var*.

**frozen**(*@Var, -Goal*)

Unify *Goal* with the goal or conjunction of goals delayed on *Var*. If no goals are frozen on *Var*,*Goal* is unified to true.

**functor**(*?Term, ?Name, ?Arity*)

True when *Term* is a term with functor *Name*/*Arity*. If *Term* is a variable it is unified with a new term whose arguments are all different variables (such a term is called a skeleton). If *Term* is atomic, *Arity* will be unified with the integer 0, and *Name* will be unified with *Term*. Raisesinstantiation\_error() if *Term* is unbound and *Name*/*Arity* is insufficiently instantiated.

SWI-Prolog also supports terms with arity 0, as in a() (see [section 5](http://www.swi-prolog.org/pldoc/man?section=extensions). Such terms must be processed using [compound\_name\_arity/3](http://www.swi-prolog.org/pldoc/man?predicate=compound_name_arity/3). The predicate [functor/3](http://www.swi-prolog.org/pldoc/man?predicate=functor/3) and [=../2](http://www.swi-prolog.org/pldoc/man?predicate=%3D../2) raise adomain\_error when faced with these terms. Without this precaution, the inconsistency demonstrated below could happen silently.85

?- functor(a(), N, A).

N = a, A = 0.

?- functor(T, a, 0).

T = a.

**garbage\_collect**

Invoke the global and trail stack garbage collector. Normally the garbage collector is invoked automatically if necessary. Explicit invocation might be useful to reduce the need for garbage collections in time-critical segments of the code. After the garbage collection [trim\_stacks/0](http://www.swi-prolog.org/pldoc/man?predicate=trim_stacks/0) is invoked to release the collected memory resources.

**garbage\_collect\_atoms**

Reclaim unused atoms. Normally invoked after [agc\_margin](http://www.swi-prolog.org/pldoc/man?section=flags#flag:agc_margin) (a Prolog flag) atoms have been created. On multithreaded versions the actual collection is delayed until there are no threads performing normal garbage collection. In this case [garbage\_collect\_atoms/0](http://www.swi-prolog.org/pldoc/man?predicate=garbage_collect_atoms/0) returns immediately. Note that there is no guarantee it will *ever* happen, as there may always be threads performing garbage collection.

**garbage\_collect\_clauses**

Reclaim retracted clauses. During normal operation, retracting a clause implies setting the*erased generation* to the current *generation* of the database and increment the generation. Keeping the clause around is both needed to realise the *logical update view* and deal with the fact that other threads may be executing the code. Both static and dynamic code is processed this way.45.

The clause garbage collector (CGC) scans the environment stacks of all threads for referenced dirty predicates and at which generation this reference accesses the predicate. It then removes the references for clauses that have been retracted before the oldest access generation from the clause list as well as the secondary clauses indexes of the predicate. If the clause list is not being scanned, the clause references and ultimately the clause itself is reclaimed.

The clause garbage collector is called under three conditions, (1) after *reloading* a source file, (2) if the memory occupied by retracted but not yet reclaimed clauses exceeds 12.5% of the program store, or (3) if skipping dead clauses in the clause lists becomes too costly.

**gen\_assoc**(*?Key, +Assoc, ?Value*)

Enumerate matching elements of *Assoc* in ascending order of their keys via backtracking.

**gen\_nb\_set**(*+Set, -Key*)

Generate all members of *Set* on backtracking in the standard order of terms. To test membership, use [add\_nb\_set/3](http://www.swi-prolog.org/pldoc/man?predicate=add_nb_set/3).

**gensym**(*+Base, -Unique*)

Generate a unique atom from base *Base* and unify it with *Unique*. *Base* should be an atom. The first call will return <*base*>1 , the next <*base*>2 , etc. Note that this is no guarantee that the atom is unique in the system.

**get**(*-Char*)

Read the current input stream and unify the next non-blank character with *Char*. *Char* is unified with -1 on end of file. The predicate [get/1](http://www.swi-prolog.org/pldoc/man?predicate=get/1) operates on character *codes*. See also[get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1).

**get**(*+Stream, -Char*)

Read the next non-blank character from *Stream*. See also [get/1](http://www.swi-prolog.org/pldoc/man?predicate=get/1), [get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1) and [get0/2](http://www.swi-prolog.org/pldoc/man?predicate=get0/2).

**get\_assoc**(*+Key, +Assoc, ?Value*)

*Value* is the value associated with *Key* in the association list *Assoc*.

**get\_assoc**(*+Key, +Assoc, ?Old, ?NewAssoc, ?New*)

*NewAssoc* is an association list identical to *Assoc* except that the value associated with *Key* is*New* instead of *Old*.

**get\_attr**(*+Var, +Module, -Value*)

Request the current *value* for the attribute named *Module*. If *Var* is not an attributed variable or the named attribute is not associated to *Var* this predicate fails silently. If *Module* is not an atom, a type error is raised.

**get\_attrs**(*+Var, -Attributes*)

Get all attributes of *Var*. *Attributes* is a term of the form att(Module, Value, MoreAttributes), where *MoreAttributes* is [] for the last attribute.

**get\_byte**(*-Byte*)

Read the current input stream and unify the next byte with *Byte* (an integer between 0 and 255). *Byte* is unified with -1 on end of file.

**get\_byte**(*+Stream, -Byte*)

Read the next byte from *Stream* and unify *Byte* with an integer between 0 and 255.

**get\_char**(*-Char*)

Read the current input stream and unify *Char* with the next character as a one-character atom. See also [atom\_chars/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2). On end-of-file, *Char* is unified to the atom end\_of\_file.

**get\_char**(*+Stream, -Char*)

Unify *Char* with the next character from *Stream* as a one-character atom. See also [get\_char/2](http://www.swi-prolog.org/pldoc/man?predicate=get_char/2),[get\_byte/2](http://www.swi-prolog.org/pldoc/man?predicate=get_byte/2) and [get\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=get_code/2).

**get\_code**(*-Code*)

Read the current input stream and unify *Code* with the character code of the next character.*Code* is unified with -1 on end of file. See also [get\_char/1](http://www.swi-prolog.org/pldoc/man?predicate=get_char/1).

**get\_code**(*+Stream, -Code*)

Read the next character code from *Stream*.

**get\_dict**(*?Key, +Dict, -Value*)

Unify the value associated with *Key* in dict with *Value*. If *Key* is unbound, all associations in *Dict*are returned on backtracking. The order in which the associations are returned is undefined. This predicate is normally accessed using the functional notation Dict.Key. See [section 5.4.1](http://www.swi-prolog.org/pldoc/man?section=dicts).

**get\_dict**(*+Key, +Dict, -Value, -NewDict, +NewValue*)

Create a new dict after updating the value for *Key*. Fails if *Value* does not unify with the current value associated with *Key*. Acts according to the following below. *Dict* is either a dict or a list the can be converted into a dict.

get\_dict(Key, Dict, Value, NewDict, NewDict) :-

get\_dict(Key, Dict, Value),

put\_dict(Key, Dict, NewDict, NewDict).

**get\_flag**(*+Key, -Value*)

True when *Value* is the value currently associated with *Key*. If *Key* does not exist, a new flag with value `0' (zero) is created.

**get\_single\_char**(*-Code*)

Get a single character from input stream `user' (regardless of the current input stream). Unlike [get\_code/1](http://www.swi-prolog.org/pldoc/man?predicate=get_code/1), this predicate does not wait for a return. The character is not echoed to the user's terminal. This predicate is meant for keyboard menu selection, etc. If SWI-Prolog was started with the **-tty** option this predicate reads an entire line of input and returns the first non-blank character on this line, or the character code of the newline (10) if the entire line consisted of blank characters.

**get\_string\_code**(*+Index, +String, -Code*)

Semi-deterministic version of [string\_code/3](http://www.swi-prolog.org/pldoc/man?predicate=string_code/3). In addition, this version provides strict range checking, throwing a domain error if *Index* is less than 1 or greater than the length of *String*. ECLiPSe provides this to support String[Index] notation.

**get\_time**(*-TimeStamp*)

Return the current time as a *TimeStamp*. The granularity is system-dependent. See [section 4.33.2.1](http://www.swi-prolog.org/pldoc/man?section=system).

**get0**(*-Char*)

Edinburgh version of the ISO [get\_code/1](http://www.swi-prolog.org/pldoc/man?predicate=get_code/1) predicate. Note that Edinburgh Prolog didn't support wide characters and therefore technically speaking [get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1) should have been mapped to [get\_byte/1](http://www.swi-prolog.org/pldoc/man?predicate=get_byte/1). The intention of [get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1), however, is to read character codes.

**get0**(*+Stream, -Char*)

Edinburgh version of the ISO [get\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=get_code/2) predicate. See also [get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1).

**getenv**(*+Name, -Value*)

Get environment variable. Fails silently if the variable does not exist. Please note that environment variable names are case-sensitive on Unix systems and case-insensitive on Windows.

**goal\_expansion**(*+Goal1, -Goal2*)

Like [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2), [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) provides for macro expansion of Prolog source code. Between [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2) and the actual compilation, the body of clauses analysed and the goals are handed to [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2), which uses the [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) hook to do user-defined expansion.

The predicate [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) is first called in the module that is being compiled, and then follows the module inheritance path as defined by [default\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=default_module/2), i.e., by default user andsystem. If *Goal* is of the form *Module*:*Goal* where *Module* is instantiated, [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) is called on *Goal* using rules from module *Module* followed by default modules for *Module*.

Only goals appearing in the body of clauses when reading a source file are expanded using this mechanism, and only if they appear literally in the clause, or as an argument to a defined meta-predicate that is annotated using `0' (see [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1)). Other cases need a real predicate definition.

The expansion hook can use [prolog\_load\_context/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_load_context/2) to obtain information about the context in which the goal is exanded such as the module, variable names or the encapsulating term.

# [goal\_expansion/4](http://www.swi-prolog.org/pldoc/consulting.html#goal_expansion/4)

**ground**(*@Term*)

True if *Term* holds no free variables.

**gdebug**

Utility defined as guitracer,debug.

**gspy**(*+Predicate*)

Utility defined as guitracer,spy(Predicate).

**gtrace**

Utility defined as guitracer,trace.

**guitracer**

Installs hooks (see [prolog\_trace\_interception/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_trace_interception/4)) into the system that redirect tracing information to a GUI front-end providing structured access to variable bindings, graphical overview of the stack and highlighting of relevant source code.

**Availability:**:- use\_module([library(gui\_tracer)](http://www.swi-prolog.org/pldoc/man?section=start-guitracer)).*(can be autoloaded)*

[Source](http://www.swi-prolog.org/pldoc/doc/home/swipl/lib/swipl/xpce/prolog/lib/gui_tracer.pl?show=src#guitracer/0)**guitracer**

This predicate installs the above-mentioned hooks that redirect tracing to the window-based environment. No window appears. The debugger window appears as actual tracing is started through [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0), by hitting a spy point defined by [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1) or a break point defined using the PceEmacs command **Prolog/Break at** (Control-c b).

**gxref**

Run cross-referencer on all currently loaded files and present a graphical overview of the result. As the predicate operates on the currently loaded application it must be run after loading the application.

**halt**

Terminate Prolog execution. This is the same as halt(0). See [halt/1](http://www.swi-prolog.org/pldoc/man?predicate=halt/1) for details.

**halt**(*+Status*)

Terminate Prolog execution with *Status*. This predicate calls [PL\_halt()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_halt) which preforms the following steps:

1. Set the Prolog flag [exit\_status](http://www.swi-prolog.org/pldoc/man?section=flags#flag:exit_status) to *Status*.
2. Call all hooks registered using [at\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=at_halt/1). If *Status* equals 0 (zero), any of these hooks calls [cancel\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=cancel_halt/1), termination is cancelled.
3. Call all hooks registered using **PL\_at\_halt()**. In the future, if any of these hooks returns non-zero, termination will be cancelled. Currently, this only prints a warning.
4. Perform the following system cleanup actions:
   * Cancel all threads, calling [thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) registered termination hooks. Threads not responding within 1 second are cancelled forcefully.
   * Flush I/O and close all streams except for standard I/O.
   * Reset the terminal if its properties were changed.
   * Remove temporary files and incomplete compilation output.
   * Reclaim memory.
5. Call exit(Status) to terminate the process

**term\_hash**(*+Term, -HashKey*)

If *Term* is a ground term (see [ground/1](http://www.swi-prolog.org/pldoc/man?predicate=ground/1)), *HashKey* is unified with a positive integer value that may be used as a hash key to the value. If *Term* is not ground, the predicate leaves *HashKey* an unbound variable. Hash keys are in the range *0 ... 16,777,215*, the maximal integer that can be stored efficiently on both 32 and 64 bit platforms.

This predicate may be used to build hash tables as well as to exploit argument indexing to find complex terms more quickly.

The hash key does not rely on temporary information like addresses of atoms and may be assumed constant over different invocations and versions of SWI-Prolog.67 Hashes differ between big and little endian machines. The [term\_hash/2](http://www.swi-prolog.org/pldoc/man?predicate=term_hash/2) predicate is cycle-safe.bug

**term\_hash**(*+Term, +Depth, +Range, -HashKey*)

As [term\_hash/2](http://www.swi-prolog.org/pldoc/man?predicate=term_hash/2), but only considers *Term* to the specified *Depth*. The top-level term has depth 1, its arguments have depth 2, etc. That is, *Depth = 0* hashes nothing; *Depth = 1* hashes atomic values or the functor and arity of a compound term, not its arguments; *Depth = 2* also indexes the immediate arguments, etc.

*HashKey* is in the range *[0 ...Range-1]*. *Range* must be in the range *[1 ... 2147483647]*

**help**

Equivalent to help([help/1](http://www.swi-prolog.org/pldoc/man?predicate=help/1)).

**help**(*+What*)

Show specified part of the manual. *What* is one of:

|  |  |
| --- | --- |
| <*Name*>/<*Arity*> | Give help on specified predicate |
| <*Name*> | Give help on named predicate with any arity or C interface function with that name |
| <*Section*> | Display specified section. Section numbers are dash-separated numbers: 2-3 refers to section 2.3 of the manual. Section numbers are obtained using [apropos/1](http://www.swi-prolog.org/pldoc/man?predicate=apropos/1). |

Examples:

|  |  |
| --- | --- |
| ?- help(assert). | Give help on predicate assert |
| ?- help(3-4). | Display section 3.4 of the manual |
| ?- help('PL\_retry'). | Give help on interface function [PL\_retry()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_retry) |

See also [apropos/1](http://www.swi-prolog.org/pldoc/man?predicate=apropos/1) and the SWI-Prolog home page at [http://www.swi-prolog.org](http://www.swi-prolog.org/), which provides a FAQ, an HTML version of the manual for online browsing, and HTML and PDF versions for downloading.

**help\_hook/1** *(hook)* User-hook in the help-system

**if**(*:Goal*)

Compile subsequent code only if *Goal* succeeds. For enhanced portability, *Goal* is processed by [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2) before execution. If an error occurs, the error is printed and processing proceeds as if *Goal* has failed.

**ignore**(*:Goal*)

Calls *Goal* as [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1), but succeeds, regardless of whether *Goal* succeeded or not. Defined as:

ignore(Goal) :-

Goal, !.

ignore(\_).

**import**(*+PredicateIndicator, ...*)

Import predicates *PredicateIndicator* into the current context module. *PredicateIndicator*must specify the source module using the <*module*>:<*pi*> construct. Note that predicates are normally imported using one of the directives use\_module/[1,2]. The [import/1](http://www.swi-prolog.org/pldoc/man?predicate=import/1) alternative is meant for handling imports into dynamically created modules. See also [export/1](http://www.swi-prolog.org/pldoc/man?predicate=export/1) and[export\_list/2](http://www.swi-prolog.org/pldoc/man?predicate=export_list/2).

**import\_module**(*+Module, -Import*)

True if *Module* inherits directly from *Import*. All normal modules only import from user, which imports from system. The predicates [add\_import\_module/3](http://www.swi-prolog.org/pldoc/man?predicate=add_import_module/3) and [delete\_import\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=delete_import_module/2)can be used to manipulate the import list. See also [default\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=default_module/2).

**in\_pce\_thread**(*:Goal*)

Assuming XPCE is running in the foreground thread, this call gives background threads the opportunity to make calls to the XPCE thread. A call to [in\_pce\_thread/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread/1) succeeds immediately, copying *Goal* to the XPCE thread. *Goal* is added to the XPCE event queue and executed synchronous to normal user events like typing and clicking.

**in\_pce\_thread\_sync**(*:Goal*)

Same as [in\_pce\_thread/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread/1), but wait for *Goal* to be completed. Success depends on the success of executing *Goal*. Variable bindings inside *Goal* are visible to the caller, but it should be noted that the values are being *copied*. If *Goal* throws an exception, this exception is re-thrown by[in\_pce\_thread/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread/1). If the calling thread is the `pce thread', [in\_pce\_thread\_sync/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread_sync/1) executes a direct meta-call. See also [pce\_thread/1](http://www.swi-prolog.org/pldoc/man?predicate=pce_thread/1).

Note that [in\_pce\_thread\_sync/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread_sync/1) is expensive because it requires copying and thread communication. For example, in\_pce\_thread\_synctrue runs at approximately 50,000 calls per second (AMD Phenom 9600B, Ubuntu 11.04).

**include**(*+File*)

Textually include the content of *File* at the position where the *directive* :- include(File).appears. The include construct is only honoured if it appears as a directive in a source file.*Textual* include (similar to C/C++ #include) is obviously useful for sharing declarations such as[dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1) or [multifile/1](http://www.swi-prolog.org/pldoc/man?predicate=multifile/1) by including a file with directives from multiple files that use these predicates.

Textually including files that contain *clauses* is less obvious. Normally, in SWI-Prolog, clauses are *owned* by the file in which they are defined. This information is used to *replace* the old definition after the file has been modified and is reloaded by, e.g., [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0). As we understand it, [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) is intended to include the same file multiple times. Including a file holding clauses multiple times into the same module is rather meaningless as it just duplicates the same clauses. Including a file holding clauses in multiple modules does not suffer from this problem, but leads to multiple equivalent *copies* of predicates. Using [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) can achieve the same result while *sharing* the predicates.

If [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) is used to load files holding clauses, and if these files are loaded only once, then these [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) directives can be replaced by other predicates (such as [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1)). However, there are several cases where either [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) has no alternative, or using any alternative also requires other changes. An example of the former is using [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) to share directives. An example of the latter are cases where clauses of different predicates are distributed over multiple files: If these files are loaded with [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1), the directive [discontiguous/1](http://www.swi-prolog.org/pldoc/man?predicate=discontiguous/1) is appropriate, whereas if they are consulted, one must use the directive [multifile/1](http://www.swi-prolog.org/pldoc/man?predicate=multifile/1).

To accommodate included files holding clauses, SWI-Prolog distinguishes between the source location of a clause (in this case the included file) and the *owner* of a clause (the file that includes the file holding the clause). The source location is used by, e.g., [edit/1](http://www.swi-prolog.org/pldoc/man?predicate=edit/1), the graphical tracer, etc., while the owner is used to determine which clauses are removed if the file is modified. Relevant information is found with the following predicates:

* [source\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file/2) describes the owner relation.
* [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) describes the source location (of the first clause).
* [clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2) provides access to both source and ownership.
* [source\_file\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file_property/2) can be used to query include relationships between files.

**initialization**(*:Goal*)

Call *Goal* *after* loading the source file in which this directive appears has been completed. In addition, *Goal* is executed if a saved state created using [qsave\_program/1](http://www.swi-prolog.org/pldoc/man?predicate=qsave_program/1) is restored.

The ISO standard only allows for using :- Term if *Term* is a *directive*. This means that arbitrary goals can only be called from a directive by means of the [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1) directive. SWI-Prolog does not enforce this rule.

The [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1) directive must be used to do program initialization in saved states (see[qsave\_program/1](http://www.swi-prolog.org/pldoc/man?predicate=qsave_program/1)). A saved state contains the predicates, Prolog flags and operators present at the moment the state was created. Other resources (records, foreign resources, etc.) must be recreated using [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1) directives or from the entry goal of the saved state.

Up to SWI-Prolog 5.7.11, *Goal* was executed immediately rather than after loading the program text in which the directive appears as dictated by the ISO standard. In many cases the exact moment of execution is irrelevant, but there are exceptions. For example,[load\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=load_foreign_library/1) must be executed immediately to make the loaded foreign predicates available for exporting. SWI-Prolog now provides the directive [use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1) to ensure immediate loading as well as loading after restoring a saved state. If the system encounters a directive :- initialization(load\_foreign\_library(...)), it will load the foreign library immediately and issue a warning to update your code. This behaviour can be extended by providing clauses for the multifile hook predicateprolog:initialize\_now(Term, Advice), where *Advice* is an atom that gives advice on how to resolve the compatibility issue.

**initialization**(*:Goal, +When*)

Similar to [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1), but allows for specifying when *Goal* is executed while loading the program text:

**now**

Execute *Goal* immediately.

**after\_load**

Execute *Goal* after loading program text. This is the same as [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1).

**restore**

Do not execute *Goal* while loading the program, but *only* when restoring a state.

**instance**(*+Reference, -Term*)

Unify *Term* with the referenced clause or database record. Unit clauses are represented as*Head* :- true.

**integer**(*@Term*)

True if *Term* is bound to an integer.

**interactor**

Create a new console and run the Prolog top level in this new console. See also[attach\_console/0](http://www.swi-prolog.org/pldoc/man?predicate=attach_console/0). In the Windows version a new interactor can also be created from the**Run/New thread** menu.

*Number* **is** *+Expr*

True when *Number* is the value to which *Expr* evaluates. Typically, [is/2](http://www.swi-prolog.org/pldoc/man?predicate=is/2) should be used with unbound left operand. If equality is to be tested, [=:=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D%3A%3D/2) should be used. For example:

|  |  |
| --- | --- |
| ?- 1 is sin(pi/2). | Fails! sin(pi/2) evaluates to the float 1.0, which does not unify with the integer 1. |
| ?- 1 =:= sin(pi/2). | Succeeds as expected. |

**is\_absolute\_file\_name**(*+File*)

True if *File* specifies an absolute path name. On Unix systems, this implies the path starts with a `/'. For Microsoft-based systems this implies the path starts with <*letter*>:. This predicate is intended to provide platform-independent checking for absolute paths. See also[absolute\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/2) and [prolog\_to\_os\_filename/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_to_os_filename/2).

**is\_assoc**(*+Assoc*)

True if Assoc is a valid association list. This predicate verifies the validity of each node in the AVL tree.

**is\_list**(*+Term*)

True if *Term* is bound to the empty list ([]) or a term with functor `'[|]''107 and arity 2 and the second argument is a list.108 This predicate acts as if defined by the definition below on*acyclic* terms. The implementation *fails* safely if *Term* represents a cyclic list.

is\_list(X) :-

var(X), !,

fail.

is\_list([]).

is\_list([\_|T]) :-

is\_list(T).

**is\_dict**(*@Term*)

True if *Term* is a dict. This is the same as is\_dict(Term,\_).

**is\_dict**(*@Term, -Tag*)

True if *Term* is a dict of *Tag*.

**is\_stream**(*+Term*)

True if *Term* is a stream name or valid stream handle. This predicate realises a safe test for the existence of a stream alias or handle.

**join\_threads**

Join all terminated threads. For normal applications, dealing with terminated threads must be part of the application logic, either detaching the thread before termination or making sure it will be joined. The predicate [join\_threads/0](http://www.swi-prolog.org/pldoc/man?predicate=join_threads/0) is intended for interactive sessions to reclaim resources from threads that died unexpectedly during development.

**keysort**(*+List, -Sorted*)

Sort a list of *pairs*. *List* must be a list of *Key*-*Value* pairs, terms whose principal functor is (-)/2.*List* is sorted on *Key* according to the standard order of terms (see [section 4.7.1](http://www.swi-prolog.org/pldoc/man?section=compare)). Duplicates are *not* removed. Sorting is *stable* with regard to the order of the *Values*, i.e., the order of multiple elements that have the same *Key* is not changed.

The [keysort/2](http://www.swi-prolog.org/pldoc/man?predicate=keysort/2) predicate is often used together with library library(pairs). It can be used to sort lists on different or multiple criteria. For example, the following predicates sorts a list of atoms according to their length, maintaining the initial order for atoms that have the same length.

:- use\_module(library(pairs)).

sort\_atoms\_by\_length(Atoms, ByLength) :-

map\_list\_to\_pairs(atom\_length, Atoms, Pairs),

keysort(Pairs, Sorted),

pairs\_values(Sorted, ByLength).

**last**(*?List, ?Last*)

Succeeds when *Last* is the last element of *List*. This predicate is semidet if *List* is a list andmulti if *List* is a partial list.

**Compatibility**

There is no de-facto standard for the argument order of [last/2](http://www.swi-prolog.org/pldoc/man?predicate=last/2). Be careful when porting code or useappend(\_, [Last], List) as a portable alternative.

**leash**(*?Ports*)

Set/query leashing (ports which allow for user interaction). *Ports* is one of *+Name*, *-Name*, *?Name* or a list of these. *+Name* enables leashing on that port, *-Name* disables it and *?Name*succeeds or fails according to the current setting. Recognised ports are call, redo, exit, failand unify. The special shorthand all refers to all ports, full refers to all ports except for the unify port (default). half refers to the call, redo and fail port.

**length**(*?List, ?Int*)

True if *Int* represents the number of elements in *List*. This predicate is a true relation and can be used to find the length of a list or produce a list (holding variables) of length *Int*. The predicate is non-deterministic, producing lists of increasing length if *List* is a *partial list* and*Int* is unbound. It raises errors if

* *Int* is bound to a non-integer.
* *Int* is a negative integer.
* *List* is neither a list nor a partial list. This error condition includes cyclic lists.109

This predicate fails if the tail of *List* is equivalent to *Int* (e.g., length(L,L)).110

**library\_directory**(*?Atom*)

Dynamic predicate used to specify library directories. Default ./lib, /lib/prolog and the system's library (in this order) are defined. The user may add library directories using[assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1), [asserta/1](http://www.swi-prolog.org/pldoc/man?predicate=asserta/1) or remove system defaults using [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1). Deprecated. New code should use [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2).

**license**(*+LicenseId*)

Intended as a directive in Prolog source files. It takes the current filename and calls [license/2](http://www.swi-prolog.org/pldoc/man?predicate=license/2).

**license**(*+LicenseId, +Component*)

Register the fact that *Component* is distributed under a license identified by *LicenseId*. The most important *LicenseId*'s are:

**swipl**

Indicates this module is distributed under the GNU General Public License (GPL) with the SWI-Prolog exception:172

*As a special exception, if you link this library with other files, compiled with SWI-Prolog, to produce an executable, this library does not by itself cause the resulting executable to be covered by the GNU General Public License. This exception does not, however, invalidate any other reasons why the executable file might be covered by the GNU General Public License.*

This should be the default for software contributed to the SWI-Prolog project as it allows the community to prosper both in the free and non-free world. Still, people using SWI-Prolog to create non-free applications must contribute sources to improvements they make to the community.

**lgpl**

This is the default license for foreign libraries linked with SWI-Prolog. Use [PL\_license()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_license)to register the condition from foreign code.

**gpl**

Indicates this module is strictly Free Software, which implies it cannot be used together with any module that is incompatible with the GPL. Please only use these conditions when forced by other code used in the component.

Other licenses known to the system are guile, gnu\_ada, x11, expat, sml, public\_domain,cryptix, bsd, zlib, lgpl\_compatible and gpl\_compatible. New licenses can be defined by adding clauses for the multifile predicate license:license/3. Below is an example. The second argument is either gpl or lgpl to indicate compatibility with these licenses. Other values cause the license to be interpreted as *proprietary*. Proprietary licenses are reported by[eval\_license/0](http://www.swi-prolog.org/pldoc/man?predicate=eval_license/0). See the file boot/license.pl for details.

:- multifile license:license/3.

license:license(mylicense, lgpl,

[ comment('My personal license'),

url('http://www.mine.org/license.html')

]).

:- license(mylicense).

**line\_count**(*+Stream, -Count*)

Unify *Count* with the number of lines read or written. Counting starts at 1.

**line\_position**(*+Stream, -Count*)

Unify *Count* with the position on the current line. Note that this assumes the position is 0 after the open. Tabs are assumed to be defined on each 8-th character, and backspaces are assumed to reduce the count by one, provided it is positive.

**list\_debug\_topics**

List currently known debug topics and their setting.

**list\_to\_assoc**(*+List, -Assoc*)

*Assoc* is an association list corresponding to the Key-Value pairs in *List*. *List* must not contain duplicate keys.

**list\_to\_set**(*+List, ?Set*)

True when *Set* has the same elements as *List* in the same order. The left-most copy of duplicate elements is retained. *List* may contain variables. Elements *E1* and *E2* are considered duplicates iff *E1* == *E2* holds. The complexity of the implementation is N\*log(N).

**Errors**

*List* is type-checked.

**author**

Ulrich Neumerkel

**See also**

[sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2) can be used to create an ordered set. Many set operations on ordered sets are order N rather than order N\*\*2. The [list\_to\_set/2](http://www.swi-prolog.org/pldoc/man?predicate=list_to_set/2) predicate is more expensive than [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2) because it involves, in addition to a sort, three linear scans of the list.

**Compatibility**

Up to version 6.3.11, [list\_to\_set/2](http://www.swi-prolog.org/pldoc/man?predicate=list_to_set/2) had complexity N\*\*2 and equality was tested using [=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D/2).

**list\_strings**

*[det]***list\_strings**(*+Options*)

List strings that appear in clauses. This predicate is used to find portability issues for changing the Prolog flag double\_quotes from codes to string, creating packed string objects. Warnings may be suppressed using the following multifile hooks:

* [**string\_predicate/1**](http://www.swi-prolog.org/pldoc/check.html#string_predicate/1) to stop checking certain predicates
* [**valid\_string\_goal/1**](http://www.swi-prolog.org/pldoc/check.html#valid_string_goal/1) to tell the checker that a goal is safe.

**See also**

Prolog flag double\_quotes.

**Availability:**:- use\_module([library(check)](http://www.swi-prolog.org/pldoc/man?section=ext-dquotes-port)).*(can be autoloaded)*

[Source](http://www.swi-prolog.org/pldoc/doc/home/swipl/lib/swipl/library/check.pl?show=src#list_strings/0)**list\_strings**

This predicate may be used to assess compatibility issues due to the representation of double quoted text as string objects. See [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings) and [section 5.2.4](http://www.swi-prolog.org/pldoc/man?section=strings). To use it, load your program into Prolog and run [list\_strings/0](http://www.swi-prolog.org/pldoc/man?predicate=list_strings/0). The predicate lists source locations of string objects encountered in the program that are not considered safe. Such string need to be examined manually, after which one of the actions below may be appropriate:

* Rewrite the code. For example, change [X] = "a" into X = 0'a.
* If a particular module relies heavily on representing strings as lists of character code, consider adding the following directive to the module. Note that this flag only applies to the module in which it appears.
* :- set\_prolog\_flag(double\_quotes, codes).
* Use a back quoted string (e.g., `text`). Note that this will not make your code run regardless of the **--traditional** command line option and code exploiting this mapping is also not portable to ISO compliant systems.
* If the strings appear in facts and usage is safe, add a clause to the multifile predicate check:string\_predicate/1 to silence [list\_strings/0](http://www.swi-prolog.org/pldoc/man?predicate=list_strings/0) on all clauses of that predicate.
* If the strings appear as an argument to a predicate that can handle string objects, add a clause to the multifile predicate check:valid\_string\_goal/1 to silence [list\_strings/0](http://www.swi-prolog.org/pldoc/man?predicate=list_strings/0).

**listing**

List all predicates from the calling module using [listing/1](http://www.swi-prolog.org/pldoc/man?predicate=listing/1). For example, ?- listing. lists clauses in the default user module and ?- lists:listing. lists the clauses in the modulelists.

**listing**(*:Pred*)

List predicates specified by *Pred*. *Pred* may be a predicate name (atom), which lists all predicates with this name, regardless of their arity. It can also be a predicate indicator (<*name*>/<*arity*> or <*name*>//<*arity*>), possibly qualified with a module. For example: ?- listing(lists:member/2)..

A listing is produced by enumerating the clauses of the predicate using [clause/2](http://www.swi-prolog.org/pldoc/man?predicate=clause/2) and printing each clause using [portray\_clause/1](http://www.swi-prolog.org/pldoc/man?predicate=portray_clause/1). This implies that the variable names are generated (*A*, *B*, ... ) and the layout is defined by rules in [portray\_clause/1](http://www.swi-prolog.org/pldoc/man?predicate=portray_clause/1).

**load\_files**(*:Files*)

Equivalent to load\_files(Files,[]). Same as [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1), See [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) for supported options.

**load\_files**(*:Files, +Options*)

The predicate [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) is the parent of all the other loading predicates except for[include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1). It currently supports a subset of the options of Quintus [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). *Files* is either a single source file or a list of source files. The specification for a source file is handed to[absolute\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/2). See this predicate for the supported expansions. *Options* is a list of options using the format *OptionName*(*OptionValue*).

The following options are currently supported:

**autoload**(*Bool*)

If true (default false), indicate that this load is a *demand* load. This implies that, depending on the setting of the Prolog flag [verbose\_autoload](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose_autoload), the load action is printed at level informational or silent. See also [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) and [current\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=current_prolog_flag/2).

**derived\_from**(*File*)

Indicate that the loaded file is derived from *File*. Used by [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0) to time-check and load the original file rather than the derived file.

**dialect**(*+Dialect*)

Load *Files* with enhanced compatibility with the target Prolog system identified by*Dialect*. See [expects\_dialect/1](http://www.swi-prolog.org/pldoc/man?predicate=expects_dialect/1) and [section C](http://www.swi-prolog.org/pldoc/man?section=dialect) for details.

**encoding**(*Encoding*)

Specify the way characters are encoded in the file. Default is taken from the Prolog flag[encoding](http://www.swi-prolog.org/pldoc/man?section=flags#flag:encoding). See [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for details.

**expand**(*Bool*)

If true, run the filenames through [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2) and load the returned files. Default is false, except for [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1) which is intended for interactive use. Flexible location of files is defined by [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2).

**format**(*+Format*)

Used to specify the file format if data is loaded from a stream using the stream(Stream)option. Default is source, loading Prolog source text. If qlf, load QLF data (see[qcompile/1](http://www.swi-prolog.org/pldoc/man?predicate=qcompile/1)).

**if**(*Condition*)

Load the file only if the specified condition is satisfied. The value true loads the file unconditionally, changed loads the file if it was not loaded before or has been modified since it was loaded the last time, and not\_loaded loads the file if it was not loaded before.

**imports**(*Import*)

Specify what to import from the loaded module. The default for [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) is all.*Import* is passed from the second argument of [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2). Traditionally it is a list of predicate indicators to import. As part of the SWI-Prolog/YAP integration, we also support *Pred* as *Name* to import a predicate under another name. Finally, *Import* can be the term except(Exceptions), where *Exceptions* is a list of predicate indicators that specify predicates that are *not* imported or *Pred* as *Name* terms to denote renamed predicates. See also [reexport/2](http://www.swi-prolog.org/pldoc/man?predicate=reexport/2) and [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2).bug

If *Import* equals all, all operators are imported as well. Otherwise, operators are *not*imported. Operators can be imported selectively by adding terms op(Pri,Assoc,Name)to the *Import* list. If such a term is encountered, all exported operators that unify with this term are imported. Typically, this construct will be used with all arguments unbound to import all operators or with only *Name* bound to import a particular operator.

**modified**(*TimeStamp*)

Claim that the source was loaded at *TimeStamp* without checking the source. This option is intended to be used together with the stream(Input) option, for example after extracting the time from an HTTP server or database.

**module**(*+Module*)

Load the indicated file into the given module, overruling the module name specified in the :- module(Name, ...) directive. This currently serves two purposes: (1) allow loading two module files that specify the same module into the same process and force and (2): force loading source code in a specific module, even if the code provides its own module name. Experimental.

**must\_be\_module**(*Bool*)

If true, raise an error if the file is not a module file. Used by use\_module/[1,2].

**qcompile**(*Atom*)

How to deal with quick-load-file compilation by [qcompile/1](http://www.swi-prolog.org/pldoc/man?predicate=qcompile/1). Values are:

**never**

Default. Do not use qcompile unless called explicitly.

**auto**

Use qcompile for all writeable files. See comment below.

**large**

Use qcompile if the file is `large'. Currently, files larger than 100 Kbytes are considered large.

**part**

If this load\_file/2 appears in a directive of a file that is compiled into Quick Load Format using [qcompile/1](http://www.swi-prolog.org/pldoc/man?predicate=qcompile/1), the contents of the argument files are included in the.qlf file instead of the loading directive.

If this option is not present, it uses the value of the Prolog flag [qcompile](http://www.swi-prolog.org/pldoc/man?section=flags#flag:qcompile) as default.

**redefine\_module**(*+Action*)

Defines what to do if a file is loaded that provides a module that is already loaded from another file. *Action* is one of false (default), which prints an error and refuses to load the file, or true, which uses [unload\_file/1](http://www.swi-prolog.org/pldoc/man?predicate=unload_file/1) on the old file and then proceeds loading the new file. Finally, there is ask, which starts interaction with the user. ask is only provided if the stream user\_input is associated with a terminal.

**reexport**(*Bool*)

If true re-export the imported predicate. Used by [reexport/1](http://www.swi-prolog.org/pldoc/man?predicate=reexport/1) and [reexport/2](http://www.swi-prolog.org/pldoc/man?predicate=reexport/2).

**register**(*Bool*)

If false, do not register the load location and options. This option is used by [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0)and [load\_hotfixes/1](http://www.swi-prolog.org/pldoc/man?predicate=load_hotfixes/1) to avoid polluting the load-context database. See[source\_file\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file_property/2).

**sandboxed**(*Bool*)

Load the file in *sandboxed* mode. This option controls the flag [sandboxed\_load](http://www.swi-prolog.org/pldoc/man?section=flags#flag:sandboxed_load). The only meaningful value for *Bool* is true. Using false while the Prolog flag is set to trueraises a permission error.

**scope\_settings**(*Bool*)

Scope [style\_check/1](http://www.swi-prolog.org/pldoc/man?predicate=style_check/1) and [expects\_dialect/1](http://www.swi-prolog.org/pldoc/man?predicate=expects_dialect/1) to the file and files loaded from the file after the directive. Default is true. The system and user initialization files (see **-f** and **-F**) are loading with scope\_settings(false).

**silent**(*Bool*)

If true, load the file without printing a message. The specified value is the default for all files loaded as a result of loading the specified files. This option writes the Prolog flag[verbose\_load](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose_load) with the negation of *Bool*.

**stream**(*Input*)

This SWI-Prolog extension compiles the data from the stream *Input*. If this option is used, *Files* must be a single atom which is used to identify the source location of the loaded clauses as well as to remove all clauses if the data is reconsulted.

This option is added to allow compiling from non-file locations such as databases, the web, the *user* (see [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1)) or other servers. It can be combined with format(qlf) to load QLF data from a stream.

The [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) predicate can be hooked to load other data or data from objects other than files. See [prolog\_load\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_load_file/2) for a description and library(http/http\_load) for an example. All hooks for [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) are documented in [section B.8](http://www.swi-prolog.org/pldoc/man?section=loadfilehook).

**load\_foreign\_library**(*:FileSpec*)

*[det]***load\_foreign\_library**(*:FileSpec, +Entry:atom*)

Load a *shared object* or *DLL*. After loading the *Entry* function is called without arguments. The default entry function is composed from =install\_=, followed by the file base-name. E.g., the load-call below calls the function install\_mylib(). If the platform prefixes extern functions with =\_=, this prefix is added before calling.

...

load\_foreign\_library(foreign(mylib)),

...

|  |  |
| --- | --- |
| ***FileSpec*** | is a specification for [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3). If searching the file fails, the plain name is passed to the OS to try the default method of the OS for locating foreign objects. The default definition of [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2) searches *<*prolog home*>*/lib/*<*arch*>* on Unix and *<*prolog home*>*/bin on Windows. |

**See also**

[use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1),2 are intended for use in directives.

# [load\_foreign\_library/2](http://www.swi-prolog.org/pldoc/foreignlink.html#load_foreign_library/2)

**locale\_create**(*-Locale, +Default, +Options*)

Create a new locale object. *Default* is either an existing locale or a string that denotes the name of a locale provided by the system, such as "en\_EN.UTF-8". The values read from the default locale can be modified using *Options*. *Options* provided are:

**alias**(*+Atom*)

Give the locale a name.

**decimal\_point**(*+Atom*)

Specify the decimal point to use.

**thousands\_sep**(*+Atom*)

Specify the string that delimits digit groups. Only effective is grouping is also specified.

**grouping**(*+List*)

Specify the grouping of digits. Groups are created from the right (least significant) digits, left of the decimal point. *List* is a list of integers, specifying the number of digits in each group, counting from the right. If the last element is repeat(Count), the remaining digits are grouped in groups of size *Count*. If the last element is a normal integer, digits further to the left are not grouped.

For example, the English locale uses

[ decimal\_point('.'), thousands\_sep(','), grouping([repeat(3)]) ]

Named locales exists until they are destroyed using [locale\_destroy/1](http://www.swi-prolog.org/pldoc/man?predicate=locale_destroy/1) and they are no longer referenced. Unnamed locales are subject to (atom) garbage collection.

**locale\_destroy**(*+Locale*)

Destroy a locale. If the locale is named, this removes the name association from the locale, after which the locale is left to be reclaimed by garbage collection.

**locale\_property**(*?Locale, ?Property*)

True when *Locale* has *Property*. Properties are the same as the *Options* described with[locale\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=locale_create/3).

**locale\_sort**(*+List, -Sorted*)

Sort a list of atoms using the current locale. *List* is a list of atoms or string objects (see [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)). *Sorted* is unified with a list containing all atoms of *List*, sorted to the rules of the current locale. See also [collation\_key/2](http://www.swi-prolog.org/pldoc/man?predicate=collation_key/2) and [setlocale/3](http://www.swi-prolog.org/pldoc/man?predicate=setlocale/3).

**make**

Consult all source files that have been changed since they were consulted. It checks *all* loaded source files: files loaded into a compiled state using pl -c ... and files loaded using[consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1) or one of its derivatives. The predicate [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0) is called after [edit/1](http://www.swi-prolog.org/pldoc/man?predicate=edit/1), automatically reloading all modified files. If the user uses an external editor (in a separate window), [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0)is normally used to update the program after editing. In addition, [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0) updates the autoload indices (see [section 2.13](http://www.swi-prolog.org/pldoc/man?section=autoload)) and runs [list\_undefined/0](http://www.swi-prolog.org/pldoc/man?predicate=list_undefined/0) from the library(check)library to report on undefined predicates.

**make\_directory**(*+Directory*)

Create a new directory (folder) on the filesystem. Raises an exception on failure. On Unix systems, the directory is created with default permissions (defined by the process *umask*setting).

**make\_library\_index**(*+Directory*)

Create an index for this directory. The index is written to the file 'INDEX.pl' in the specified directory. Fails with a warning if the directory does not exist or is write protected.

**make\_library\_index**(*+Directory, +ListOfPatterns*)

Normally used in MKINDEX.pl, this predicate creates INDEX.pl for *Directory*, indexing all files that match one of the file patterns in *ListOfPatterns*.

Sometimes library packages consist of one public load file and a number of files used by this load file, exporting predicates that should not be used directly by the end user. Such a library can be placed in a sub-directory of the library and the files containing public functionality can be added to the index of the library. As an example we give the XPCE library's MKINDEX.pl, including the public functionality of trace/browse.pl to the autoloadable predicates for the XPCE package.

:- make\_library\_index('.',

[ '\*.pl',

'trace/browse.pl'

]).

**map\_assoc**(*:Goal, +Assoc*)

*Goal(V)* is true for every value V in *Assoc*.

**map\_assoc**(*:Goal, +AssocIn, ?AssocOut*)

*AssocOut* is *AssocIn* with *Goal* applied to all corresponding pairs of values.

**dict\_create**(*-Dict, +Tag, +Data*)

Create a dict in *Tag* from *Data*. *Data* is a list of attribute-value pairs using the syntaxKey:Value, Key=Value, Key-Value or Key(Value). An exception is raised if *Data* is not a proper list, one of the elements is not of the shape above, a key is neither an atom nor a small integer or there is a duplicate key.

**dict\_pairs**(*?Dict, ?Tag, ?Pairs*)

Bi-directional mapping between a dict and an ordered list of pairs (see [section A.20](http://www.swi-prolog.org/pldoc/man?section=pairs)).

**max\_assoc**(*+Assoc, ?Key, ?Value*)

*Key* and *Value* are key and value of the element with the largest key in *Assoc*.

**memberchk**(*?Elem, +List*)

True when *Elem* is an element of *List*. This `chk' variant of [member/2](http://www.swi-prolog.org/pldoc/man?predicate=member/2) is semi deterministic and typically used to test membership of a list. Raises a type error if scanning *List* encounters a non-list. Note that [memberchk/2](http://www.swi-prolog.org/pldoc/man?predicate=memberchk/2) does *not* perform a full list typecheck. For example,memberchk(a, [a|b]) succeeds without error and [memberchk/2](http://www.swi-prolog.org/pldoc/man?predicate=memberchk/2) loops on a cyclic list if *Elem*is not a member of *List*.

**message\_hook**(*+Term, +Kind, +Lines*)

Hook predicate that may be defined in the module user to intercept messages from[print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). *Term* and *Kind* are the same as passed to [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). *Lines* is a list of format statements as described with [print\_message\_lines/3](http://www.swi-prolog.org/pldoc/man?predicate=print_message_lines/3). See also [message\_to\_string/2](http://www.swi-prolog.org/pldoc/man?predicate=message_to_string/2).

This predicate must be defined dynamic and multifile to allow other modules defining clauses for it too.

**message\_line\_element***(+Stream, +Term)* is **semidet***[multifile,* [*library(ansi\_term)*](http://www.swi-prolog.org/pldoc/doc/swi/library/ansi_term.pl)*]*

Hook implementation that deals with ansi(+Attr, +Fmt, +Args) in message specifications.

**message\_property**(*+Kind, ?Property*)

This hook can be used to define additional message kinds and the way they are displayed. The following properties are defined:

**color**(*-Attributes*)

Print message using ANSI terminal attributes. See [ansi\_format/3](http://www.swi-prolog.org/pldoc/man?predicate=ansi_format/3) for details. Here is an example, printing help messages in blue:

:- multifile user:message\_property/2.

user:message\_property(help, color([fg(blue)])).

**prefix**(*-Prefix*)

Prefix printed before each line. This argument is handed to [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3). The default is'~N'. For example, messages of kind warning use '~NWarning: '.

**location\_prefix**(*+Location, -FirstPrefix, -ContinuePrefix*)

Used for printing messages that are related to a source location. Currently, *Location* is a term *File*:*Line*. *FirstPrefix* is the prefix for the first line and *-ContinuePrefix* is the prefix for continuation lines. For example, the default for errors is

location\_prefix(File:Line,

'~NERROR: ~w:~d:'-[File,Line], '~N\t')).

**stream**(*-Stream*)

Stream to which to print the message. Default is user\_error.

**wait**(*-Seconds*)

Amount of time to wait after printing the message. Default is not to wait.

**message\_queue\_create**(*?Queue*)

Equivalent to message\_queue\_create(Queue,[]). For compatibility, callingmessage\_queue\_create(+Atom) is equivalent to message\_queue\_create(Queue, [alias(Atom)]). New code should use [message\_queue\_create/2](http://www.swi-prolog.org/pldoc/man?predicate=message_queue_create/2) to create a named queue.

**message\_queue\_create**(*-Queue, +Options*)

Create a message queue from *Options*. Defined options are:

**alias**(*+Alias*)

Create a message queue that is identified by the atom *Alias*. Message queues created this way must be explicitly destroyed by the user. If the alias option is omitted, an*Anonymous* queue is created that is indentified by a *blob* (see [section 10.4.7](http://www.swi-prolog.org/pldoc/man?section=foreigninclude)) and subject to garbage collection.149

**max\_size**(*+Size*)

Maximum number of terms in the queue. If this number is reached,[thread\_send\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/2) will suspend until the queue is drained. The option can be used if the source, sending messages to the queue, is faster than the drain, consuming the messages.

**message\_queue\_destroy**(*+Queue*)

Destroy a message queue created with [message\_queue\_create/1](http://www.swi-prolog.org/pldoc/man?predicate=message_queue_create/1). A permission error is raised if *Queue* refers to (the default queue of) a thread. Other threads that are waiting for *Queue*using [thread\_get\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/2) receive an existence error.

**message\_queue\_property**(*?Queue, ?Property*)

True if *Property* is a property of *Queue*. Defined properties are:

**alias**(*Alias*)

Queue has the given alias name.

**max\_size**(*Size*)

Maximum number of terms that can be in the queue. See [message\_queue\_create/2](http://www.swi-prolog.org/pldoc/man?predicate=message_queue_create/2). This property is not present if there is no limit (default).

**size**(*Size*)

Queue currently contains *Size* terms. Note that due to concurrent access the returned value may be outdated before it is returned. It can be used for debugging purposes as well as work distribution purposes.

The size(Size) property is always present and may be used to enumerate the created message queues. Note that this predicate does *not enumerate* threads, but can be used to query the properties of the default queue of a thread.

**message\_to\_string**(*+Term, -String*)

Translates a message term into a string object (see [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)).

**meta\_predicate** *+Head, ...*

Define the predicates referenced by the comma-separated list *Head* as *meta-predicates*. Each argument of each head is a *meta argument specifier*. Defined specifiers are given below. Only 0..9, : and ^ are interpreted; the mode declarations +, - and ? are ignored.

**0..9**

The argument is a term that is used to reference a predicate with *N* more arguments than the given argument term. For example: call(0) or maplist(1, +).

**:**

The argument is module-sensitive, but does not directly refer to a predicate. For example: consult(:).

**-**

The argument is not module-sensitive and unbound on entry.

**?**

The argument is not module-sensitive and the mode is unspecified.

**\***

The argument is not module-sensitive and the mode is unspecified. The specification \*is equivalent to ?. It is accepted for compatibility reasons. The predicate[predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) reports arguments declared using \* with ?.

**+**

The argument is not module-sensitive and bound (i.e., nonvar) on entry.

**^**

This extension is used to denote the possibly ^-annotated goal of [setof/3](http://www.swi-prolog.org/pldoc/man?predicate=setof/3), [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3),[aggregate/3](http://www.swi-prolog.org/pldoc/man?predicate=aggregate/3) and [aggregate/4](http://www.swi-prolog.org/pldoc/man?predicate=aggregate/4). It is processed similar to `0', but leaving the ^/2 intact.

**//**

The argument is a DCG body. See [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3).

Each argument that is module-sensitive (i.e., marked 0..9, : or ^) is qualified with the context module of the caller if it is not already qualified. The implementation ensures that the argument is passed as <*module*>:<*term*>, where <*module*> is an atom denoting the name of a module and <*term*> itself is not a :/2 term where the first argument is an atom. Below is a simple declaration and a number of queries.

:- meta\_predicate

meta(0, +).

meta(Module:Term, \_Arg) :-

format('Module=~w, Term = ~q~n', [Module, Term]).

?- meta(test, x).

Module=user, Term = test

?- meta(m1:test, x).

Module=m1, Term = test

?- m2:meta(test, x).

Module=m2, Term = test

?- m1:meta(m2:test, x).

Module=m2, Term = test

?- meta(m1:m2:test, x).

Module=m2, Term = test

?- meta(m1:42:test, x).

Module=42, Term = test

The [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1) declaration is the portable mechanism for defining meta-predicates and replaces the old SWI-Prolog specific mechanism provided by the deprecated predicates[module\_transparent/1](http://www.swi-prolog.org/pldoc/man?predicate=module_transparent/1), [context\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=context_module/1) and [strip\_module/3](http://www.swi-prolog.org/pldoc/man?predicate=strip_module/3). See also [section 6.15](http://www.swi-prolog.org/pldoc/man?section=modulecompat).

**min\_assoc**(*+Assoc, ?Key, ?Value*)

*Key* and *Value* are key and value of the element with the smallest key in *Assoc*.

**module**(*+Module*)

The call module(*Module*) may be used to switch the default working module for the interactive top level (see [prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0)). This may be used when debugging a module. The example below lists the clauses of file\_of\_label/2 in the module tex.

1 ?- module(tex).

true.

tex: 2 ?- listing(file\_of\_label/2).

...

**module**(*+Module, +PublicList*)

This directive can only be used as the first term of a source file. It declares the file to be a*module file*, defining a module named *Module*. Note that a module name is an atom. The module exports the predicates of *PublicList*. *PublicList* is a list of predicate indicators (name/arity or name//arity pairs) or operator declarations using the format op(Precedence, Type, Name). Operators defined in the export list are available inside the module as well as to modules importing this module. See also [section 4.24](http://www.swi-prolog.org/pldoc/man?section=operators).

Compatible to Ciao Prolog, if *Module* is unbound, it is unified with the basename without extension of the file being loaded.

**module**(*+Module, +PublicList, +Dialect*)

Same as [module/2](http://www.swi-prolog.org/pldoc/man?predicate=module/2). The additional *Dialect* argument provides a list of *language options*. Each atom in the list *Dialect* is mapped to a [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) goal as given below. See also [section C](http://www.swi-prolog.org/pldoc/man?section=dialect). The third argument is supported for compatibility with the [Prolog Commons project](http://prolog-commons.org/).

:- use\_module(library(dialect/LangOption)).

**module\_property**(*?Module, ?Property*)

True if *Property* is a property of *Module*. Defined properties are:

**class**(*-Class*)

True when *Class* is the class of the module. Defined classes are

**user**

Default for user-defined modules.

**system**

Module system and modules from <*home*>/boot.

**library**

Other modules from the system directories.

**temporary**

Module is temporary.

**test**

Modules that create tests.

**development**

Modules that only support the development environment.

**file**(*?File*)

True if *Module* was loaded from *File*.

**line\_count**(*-Line*)

True if *Module* was loaded from the N-th line of file.

**exports**(*-ListOfPredicateIndicators*)

True if *Module* exports the given predicates. Predicate indicators are in canonical form (i.e., always using name/arity and never the DCG form name//arity). Future versions may also use the DCG form and include public operators. See also [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2).

**exported\_operators**(*-ListOfOperators*)

True if *Module* exports the given operators. Each exported operator is represented as a term op(Pri,Assoc,Name).

**program\_size**(*-Bytes*)

Memory (in bytes) used for string the predicates of this module. This figure includes the predicate header and clauses. Future versions might give a more precise number, including e.g., the clause index tables.

**program\_space**(*-Bytes*)

If present, this number limits the program\_size. See [set\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=set_module/1).

**module\_transparent**(*+Preds*)

*Preds* is a comma-separated list of name/arity pairs (like [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1)). Each goal associated with a transparent-declared predicate will inherit the *context module* from its parent goal.

**msort**(*+List, -Sorted*)

Equivalent to [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2), but does not remove duplicates. Raises a type\_error if *List* is a cyclic list or not a list.

**multifile** *:PredicateIndicator, ...*

Informs the system that the specified predicate(s) may be defined over more than one file. This stops [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1) from redefining a predicate when a new definition is found.

**mutex\_create**(*?MutexId*)

Create a mutex. If *MutexId* is an atom, a *named* mutex is created. If it is a variable, an anonymous mutex reference is returned. Anonymous mutexes are subject to (atom) garbage collection.

**mutex\_create**(*-MutexId, +Options*)

Create a mutex using options. Defined options are:

**alias**(*Alias*)

Set the alias name. Using mutex\_create(X, [alias(name)]) is preferred over the equivalent mutex\_create(name).

**mutex\_destroy**(*+MutexId*)

Destroy a mutex. If the mutex is not locked, it is destroyed and further access yields anexistence\_error exception. As of version 7.1.19, this behaviour is reliable. If the mutex is locked, the mutex is sheduled for *delayed destruction*: it will be destroyed when it becomes unlocked.

**mutex\_lock**(*+MutexId*)

Lock the mutex. Prolog mutexes are *recursive* mutexes: they can be locked multiple times by the same thread. Only after unlocking it as many times as it is locked does the mutex become available for locking by other threads. If another thread has locked the mutex the calling thread is suspended until the mutex is unlocked.

If *MutexId* is an atom, and there is no current mutex with that name, the mutex is created automatically using [mutex\_create/1](http://www.swi-prolog.org/pldoc/man?predicate=mutex_create/1). This implies named mutexes need not be declared explicitly.

Please note that locking and unlocking mutexes should be paired carefully. Especially make sure to unlock mutexes even if the protected code fails or raises an exception. For most common cases, use [with\_mutex/2](http://www.swi-prolog.org/pldoc/man?predicate=with_mutex/2), which provides a safer way for handling Prolog-level mutexes. The predicate [setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3) is another way to guarantee that the mutex is unlocked while retaining non-determinism.

**mutex\_property**(*?MutexId, ?Property*)

True if *Property* is a property of *MutexId*. Defined properties are:

**alias**(*Alias*)

Mutex has the defined alias name. See [mutex\_create/2](http://www.swi-prolog.org/pldoc/man?predicate=mutex_create/2) using the `alias' option.

**status**(*Status*)

Current status of the mutex. One of unlocked if the mutex is currently not locked, orlocked(Owner, Count) if mutex is locked *Count* times by thread *Owner*. Note that unless *Owner* is the calling thread, the locked status can change at any time. There is no useful application of this property, except for diagnostic purposes.bug

**mutex\_statistics**

Print usage statistics on internal mutexes and mutexes associated with dynamic predicates. For each mutex two numbers are printed: the number of times the mutex was acquired and the number of *collisions*: the number of times the calling thread has to wait for the mutex. Generally collision count is close to zero on single-CPU hardware.

**mutex\_trylock**(*+MutexId*)

As [mutex\_lock/1](http://www.swi-prolog.org/pldoc/man?predicate=mutex_lock/1), but if the mutex is held by another thread, this predicates fails immediately.

**mutex\_unlock**(*+MutexId*)

Unlock the mutex. This can only be called if the mutex is held by the calling thread. If this is not the case, a permission\_error exception is raised.

**mutex\_unlock\_all**

Unlock all mutexes held by the current thread. This predicate should not be needed if mutex unlocking is guaranteed with [with\_mutex/2](http://www.swi-prolog.org/pldoc/man?predicate=with_mutex/2) or [setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3).151

**name**(*?Atomic, ?CodeList*)

*CodeList* is a list of character codes representing the same text as *Atomic*. Each of the arguments may be a variable, but not both. When *CodeList* describes an integer or floating point number and *Atomic* is a variable, *Atomic* will be unified with the numeric value described by *CodeList* (e.g., name(N, "300"), 400 is N + 100 succeeds). If *CodeList* is not a representation of a number, *Atomic* will be unified with the atom with the name given by the character code list. When *Atomic* is an atom or number, the unquoted print representation of it as a character code list will be unified with *CodeList*.

Note that it is not possible to produce the atom '300' using [name/2](http://www.swi-prolog.org/pldoc/man?predicate=name/2), and that name(300, CodeList), name('300', CodeList) succeeds. For these reasons, new code should consider using the ISO predicates [atom\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_codes/2) or [number\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=number_codes/2).93 See also[atom\_number/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_number/2).

**nb\_current**(*?Name, ?Value*)

Enumerate all defined variables with their value. The order of enumeration is undefined. Note that [nb\_current/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_current/2) can be used as an alternative for [nb\_getval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_getval/2) to request the value of a variable and fail silently if the variable does not exists.

**nb\_delete**(*+Name*)

Delete the named global variable. Succeeds also if the named variable does not exist.

**nb\_getval**(*+Name, -Value*)

The [nb\_getval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_getval/2) predicate is a synonym for [b\_getval/2](http://www.swi-prolog.org/pldoc/man?predicate=b_getval/2), introduced for compatibility and symmetry. As most scenarios will use a particular global variable using either non-backtrackable or backtrackable assignment, using [nb\_getval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_getval/2) can be used to document that the variable is non-backtrackable. Raises existence\_error(variable, Name) if the variable does not exist.

**nb\_linkarg**(*+Arg, +Term, +Value*)

As [nb\_setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_setarg/3), but like [nb\_linkval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_linkval/2) it does *not* duplicate *Value*. Use with extreme care and consult the documentation of [nb\_linkval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_linkval/2) before use.

**nb\_linkval**(*+Name, +Value*)

Associates the term *Value* with the atom *Name* without copying it. This is a fast special-purpose variation of [nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2) intended for expert users only because the semantics on backtracking to a point before creating the link are poorly defined for compound terms. The principal term is always left untouched, but backtracking behaviour on arguments is undone if the original assignment was *trailed* and left alone otherwise, which implies that the history that created the term affects the behaviour on backtracking. Consider the following example:

demo\_nb\_linkval :-

T = nice(N),

( N = world,

nb\_linkval(myvar, T),

fail

; nb\_getval(myvar, V),

writeln(V)

).

**nb\_set\_to\_list**(*+Set, -List*)

Unify *List* with a list of all elements in *Set* in the standard order of terms (i.e., an *ordered list*).

**nb\_set\_dict**(*+Key, !Dict, +Value*)

Destructively update the value associated with *Key* in *Dict* to a copy of *Value*. The update is *not*undone on backtracking. This predicate raises an existence error if *Key* does not appear in*Dict*. The update semantics are equivalent to [nb\_setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_setarg/3) and [nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2).

**nb\_setarg**(*+Arg, +Term, +Value*)

Assigns the *Arg*-th argument of the compound term *Term* with the given *Value* as [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3), but on backtracking the assignment is *not* reversed. If *Value* is not atomic, it is duplicated using[duplicate\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=duplicate_term/2). This predicate uses the same technique as [nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2). We therefore refer to the description of [nb\_setval/2](http://www.swi-prolog.org/pldoc/man?predicate=nb_setval/2) for details on non-backtrackable assignment of terms. This predicate is compatible with GNU-Prolog setarg(A,T,V,false), removing the type restriction on *Value*. See also [nb\_linkarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_linkarg/3). Below is an example for counting the number of solutions of a goal. Note that this implementation is thread-safe, reentrant and capable of handling exceptions. Realising these features with a traditional implementation based on assert/retract or [flag/3](http://www.swi-prolog.org/pldoc/man?predicate=flag/3) is much more complicated.

:- meta\_predicate

succeeds\_n\_times(0, -).

succeeds\_n\_times(Goal, Times) :-

Counter = counter(0),

( Goal,

arg(1, Counter, N0),

N is N0 + 1,

nb\_setarg(1, Counter, N),

fail

; arg(1, Counter, Times)

).

**nb\_setval**(*+Name, +Value*)

Associates a copy of *Value* created with [duplicate\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=duplicate_term/2) with the atom *Name*. Note that this can be used to set an initial value other than [] prior to backtrackable assignment.

**nl**

Write a newline character to the current output stream. On Unix systems [nl/0](http://www.swi-prolog.org/pldoc/man?predicate=nl/0) is equivalent toput(10).

**nl**(*+Stream*)

Write a newline to *Stream*.

**nodebug**

Stop debugger. Implemented by the Prolog flag [debug](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debug). See also [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0).

[**nodebug/1**](http://www.swi-prolog.org/pldoc/debug.html#nodebug/1) Disable debug-topic

**noguitracer**

Revert back to the textual tracer.

**Availability:**:- use\_module([library(gui\_tracer)](http://www.swi-prolog.org/pldoc/man?section=start-guitracer)).*(can be autoloaded)*

[Source](http://www.swi-prolog.org/pldoc/doc/home/swipl/lib/swipl/xpce/prolog/lib/gui_tracer.pl?show=src#noguitracer/0)**noguitracer**

Disable the hooks installed by [guitracer/0](http://www.swi-prolog.org/pldoc/man?predicate=guitracer/0), reverting to normal text console-based tracing.

**nonvar**(*@Term*)

True if *Term* currently is not a free variable.

**noprofile**(*+Name/+Arity, ...*)

Declares the predicate *Name*/*Arity* to be invisible to the profiler. The time spent in the named predicate is added to the caller, and the callees are linked directly to the caller. This is particularly useful for simple meta-predicates such as [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1), [ignore/1](http://www.swi-prolog.org/pldoc/man?predicate=ignore/1), [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3), etc.

**noprotocol**

Stop making a protocol of the user interaction. Pending output is flushed on the file.

**normalize\_space**(*-Out, +In*)

Normalize white space in *In*. All leading and trailing white space is removed. All non-empty sequences for Unicode white space characters are replaced by a single space (\u0020) character. *Out* uses the same conventions as [with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2) and [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3).

**nospy**(*+Pred*)

Remove spy point from all predicates meeting the predicate specification *Pred*.

**nospyall**

Remove all spy points from the entire program.

**not**(*:Goal*)

True if *Goal* cannot be proven. Retained for compatibility only. New code should use [\+/1](http://www.swi-prolog.org/pldoc/man?predicate=%5C%2B/1).

**notrace**

Stop the tracer. [notrace/0](http://www.swi-prolog.org/pldoc/man?predicate=notrace/0) itself cannot be seen in the tracer.

**notrace**(*:Goal*)

Call *Goal*, but suspend the debugger while *Goal* is executing. The current implementation cuts the choice points of *Goal* after successful completion. See [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1). Later implementations may have the same semantics as [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1).

**nth\_clause**(*?Pred, ?Index, ?Reference*)

Provides access to the clauses of a predicate using their index number. Counting starts at 1. If*Reference* is specified it unifies *Pred* with the most general term with the same name/arity as the predicate and *Index* with the index number of the clause. Otherwise the name and arity of*Pred* are used to determine the predicate. If *Index* is provided, *Reference* will be unified with the clause reference. If *Index* is unbound, backtracking will yield both the indexes and the references of all clauses of the predicate. The following example finds the 2nd clause of[append/3](http://www.swi-prolog.org/pldoc/man?predicate=append/3):

?- use\_module(library(lists)).

...

?- nth\_clause(append(\_,\_,\_), 2, Ref), clause(Head, Body, Ref).

Ref = <clause>(0x994290),

Head = lists:append([\_G23|\_G24], \_G21, [\_G23|\_G27]),

Body = append(\_G24, \_G21, \_G27).

**nth\_integer\_root\_and\_remainder**(*+N, +I, -Root, -Remainder*)

True when *Root \*\* N + Remainder = I*. *N* and *I* must be integers.98 *N* must be one or more. If *I* is negative and *N* is *odd*, *Root* and *Remainder* are negative, i.e., the following holds for *I < 0*:

% I < 0,

% N mod 2 =\= 0,

nth\_integer\_root\_and\_remainder(

N, I, Root, Remainder),

IPos is -I,

nth\_integer\_root\_and\_remainder(

N, IPos, RootPos, RemainderPos),

Root =:= -RootPos,

Remainder =:= -RemainderPos.

**number**(*@Term*)

True if *Term* is bound to an integer or floating point number.47

**number\_chars**(*?Number, ?CharList*)

Similar to [atom\_chars/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2), but converts between a number and its representation as a list of one-character atoms. Fails with a syntax\_error if *Number* is unbound or *CharList* does not describe a number. Following the ISO standard, it allows for *leading* white space (including newlines) and does not allow for *trailing* white space.91

**number\_codes**(*?Number, ?CodeList*)

As [number\_chars/2](http://www.swi-prolog.org/pldoc/man?predicate=number_chars/2), but converts to a list of character codes rather than one-character atoms. In the mode (-, +), both predicates behave identically to improve handling of non-ISO source.

**number\_string**(*?Number, ?String*)

Bi-directional conversion between a number and a string. At least one of the two arguments must be instantiated. Besides the type used to represent the text, this predicate differs in several ways from its ISO cousin:130

* If *String* does not represent a number, the predicate *fails* rather than throwing a syntax error exception.
* Leading white space and Prolog comments are *not* allowed.
* Numbers may start with '+' or '-'.
* It is *not* allowed to have white space between a leading '+' or '-' and the number.
* Floating point numbers in exponential notation do not require a dot before exponent, i.e., "1e10" is a valid number.
* **numbervars**(*+Term, +Start, -End*)
* Unify the free variables in *Term* with a term $VAR(N), where *N* is the number of the variable. Counting starts at *Start*. *End* is unified with the number that should be given to the next variable.bugOnly *tagged integers* are supported (see the Prolog flag [max\_tagged\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:max_tagged_integer)). This suffices to count all variables that can appear in the largest term that can be represented, but does not support arbitrary large integer values for *Start*. On overflow, a representation\_error(tagged\_integer) exception is raised. The example below illustrates this. Note that the toplevel prints '$VAR'(0) as *A*due to the numbervars(true) option used to print answers.
* ?- Term = f(X,Y,X),
* numbervars(Term, 0, End),
* write\_canonical(Term), nl.
* f('$VAR'(0),'$VAR'(1),'$VAR'(0))
* Term = f(A, B, A),
* X = A,
* Y = B,
* End = 2.

See also the numbervars option to [write\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=write_term/3) and [numbervars/4](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/4).

**numbervars**(*+Term, +Start, -End, +Options*)

As [numbervars/3](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/3), providing the following options:

**functor\_name**(*+Atom*)

Name of the functor to use instead of $VAR.

**attvar**(*+Action*)

What to do if an attributed variable is encountered. Options are skip, which causes[numbervars/3](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/3) to ignore the attributed variable, bind which causes it to thread it as a normal variable and assign the next '$VAR'(N) term to it, or (default) error which raises a type\_error exception.87

**singletons**(*+Bool*)

If true (default false), [numbervars/4](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/4) does singleton detection. Singleton variables are unified with '$VAR'('\_'), causing them to be printed as \_ by [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) using the numbervars option. This option is exploited by [portray\_clause/2](http://www.swi-prolog.org/pldoc/man?predicate=portray_clause/2) and[write\_canonical/2](http://www.swi-prolog.org/pldoc/man?predicate=write_canonical/2).bug

**on\_signal**(*+Signal, -Old, :New*)

Determines the reaction on *Signal*. *Old* is unified with the old behaviour, while the behaviour is switched to *New*. As with similar environment control predicates, the current value is retrieved using on\_signal(Signal, Current, Current).

The action description is an atom denoting the name of the predicate that will be called if*Signal* arrives. [on\_signal/3](http://www.swi-prolog.org/pldoc/man?predicate=on_signal/3) is a meta-predicate, which implies that <*Module*>:<*Name*> refers to <*Name*>/1 in module <*Module*>. The handler is called with a single argument: the name of the signal as an atom. The Prolog names for signals are explained below.

Two predicate names have special meaning. throw implies Prolog will map the signal onto a Prolog exception as described in [section 4.10](http://www.swi-prolog.org/pldoc/man?section=exception). default resets the handler to the settings active before SWI-Prolog manipulated the handler.

Signals bound to a foreign function through [PL\_signal()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_signal) are reported using the term$foreign\_function(Address).

After receiving a signal mapped to throw, the exception raised has the following structure:

error(signal(<SigName>, <SigNum>), <*Context*>)

The signal names are defined by the POSIX standard as symbols of the form SIG<SIGNAME>. The Prolog name for a signal is the lowercase version of <SIGNAME>. The predicate[current\_signal/3](http://www.swi-prolog.org/pldoc/man?predicate=current_signal/3) may be used to map between names and signals.

Initially, some signals are mapped to throw, while all other signals are default. The following signals throw an exception: fpe, alrm, xcpu, xfsz and vtalrm.

**once**(*:Goal*)

Make a possibly *nondet* goal *semidet*, i.e., succeed at most once. Defined as:

once(Goal) :-

call(Goal), !.

[once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1) can in many cases be replaced with [->/2](http://www.swi-prolog.org/pldoc/man?predicate=send_arrow/2). The only difference is how the cut behaves (see !/0). The following two clauses below are identical. Be careful about the interaction with[;/2](http://www.swi-prolog.org/pldoc/man?predicate=%3B/2). The library(apply\_macros) library defines an inline expansion of [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1), mapping it to(Goal\send{true};fail). Using the full if-then-else constructs prevents its semantics from being changed when embedded in a [;/2](http://www.swi-prolog.org/pldoc/man?predicate=%3B/2) disjunction.

1) a :- once((b, c)), d.

2) a :- b, c -> d.

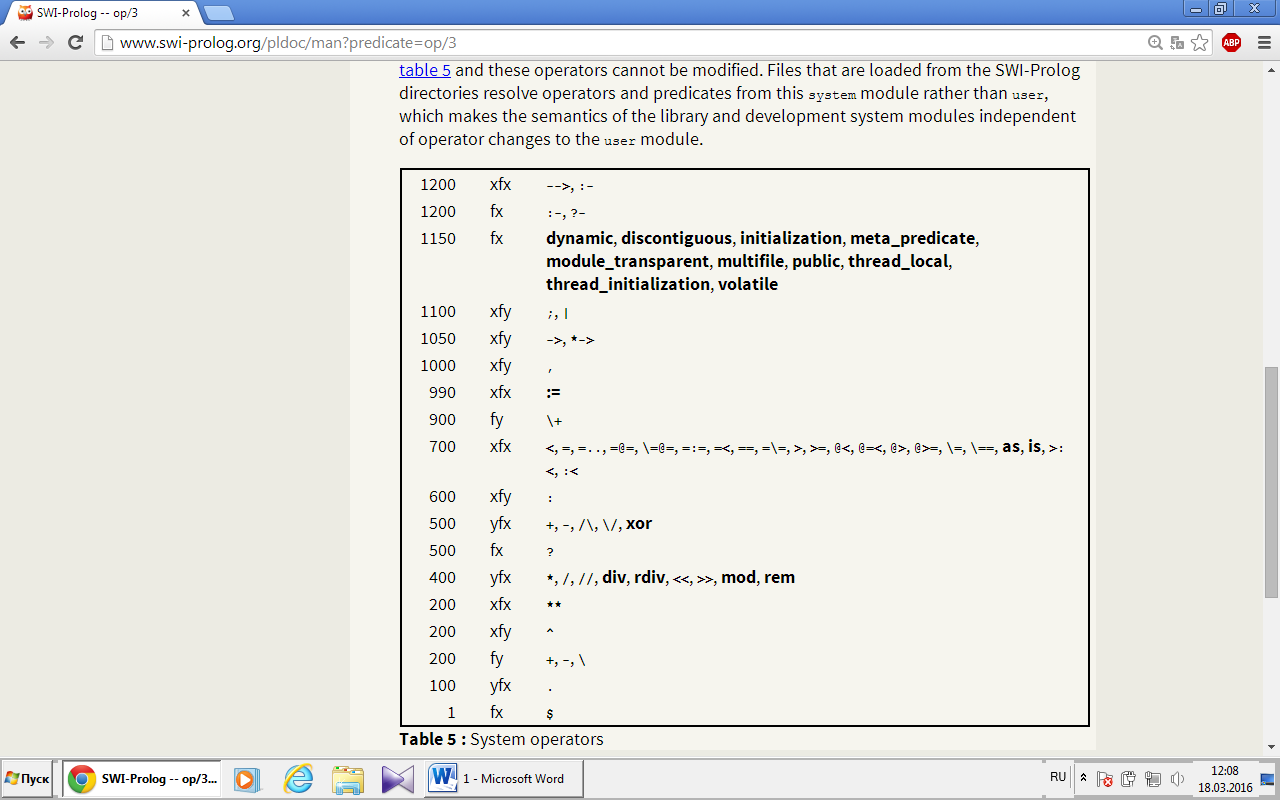
**op**(*+Precedence, +Type, :Name*)

Declare *Name* to be an operator of type *Type* with precedence *Precedence*. *Name* can also be a list of names, in which case all elements of the list are declared to be identical operators.*Precedence* is an integer between 0 and 1200. Precedence 0 removes the declaration. *Type* is one of: xf, yf, xfx, xfy, yfx, fy or fx. The `f' indicates the position of the functor, while x andy indicate the position of the arguments. `y' should be interpreted as ``on this position a term with precedence lower or equal to the precedence of the functor should occur''. For `x' the precedence of the argument must be strictly lower. The precedence of a term is 0, unless its principal functor is an operator, in which case the precedence is the precedence of this operator. A term enclosed in parentheses ( ... ) has precedence 0.

The predefined operators are shown in [table 5](http://www.swi-prolog.org/pldoc/operators.html#tab:operators). Operators can be redefined, unless prohibited by one of the limitations below. Applications must be careful with (re-)defining operators because changing operators may cause (other) files to be interpreted **differently**. Often this will lead to a syntax error. In other cases, text is read silently into a different term which may lead to subtle and difficult to track errors.

* It is not allowed to redefine the comma (',').
* The bar (|) can only be (re-)defined as infix operator with priority not less than 1001.
* It is not allowed to define the empty list ([]) or the curly-bracket pair ({}) as operators.

In SWI-Prolog, operators are *local* to a module (see also [section 6.8](http://www.swi-prolog.org/pldoc/man?section=moduleop)). Keeping operators in modules and using controlled import/export of operators as described with the [module/2](http://www.swi-prolog.org/pldoc/man?predicate=module/2)directive keep the issues manageable. The module system provides the operators from [table 5](http://www.swi-prolog.org/pldoc/operators.html#tab:operators) and these operators cannot be modified. Files that are loaded from the SWI-Prolog directories resolve operators and predicates from this system module rather than user, which makes the semantics of the library and development system modules independent of operator changes to the user module.



**Table 5 :** System operators

**open**(*+SrcDest, +Mode, --Stream*)

Equivalent to [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) with an empty option list.

**open**(*+SrcDest, +Mode, --Stream, +Options*)

True when *SrcDest* can be opened in *Mode* and *Stream* is an I/O stream to/from the object.*SrcDest* is normally the name of a file, represented as an atom or string. *Mode* is one of read,write, append or update. Mode append opens the file for writing, positioning the file pointer at the end. Mode update opens the file for writing, positioning the file pointer at the beginning of the file without truncating the file. *Stream* is either a variable, in which case it is bound to an integer identifying the stream, or an atom, in which case this atom will be the stream identifier.72

SWI-Prolog also allows *SrcDest* to be a term pipe(Command). In this form, *Command* is started as a child process and if *Mode* is write, output written to *Stream* is sent to the standard input of *Command*. Viso versa, if *Mode* is read, data written by *Command* to the standard output may be read from *Stream*. On Unix systems, *Command* is handed to popen() which hands it to the Unix shell. On Windows, *Command* is executed directly. See also [process\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=process_create/3) fromlibrary(process).

The following *Options* are recognised by [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4):

**alias**(*Atom*)

Gives the stream a name. Below is an example. Be careful with this option as stream names are global. See also [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2).

?- open(data, read, Fd, [alias(input)]).

...,

read(input, Term),

...

**bom**(*Bool*)

Check for a BOM (*Byte Order Marker*) or write one. If omitted, the default is true for mode read and false for mode write. See also [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) and especially[section 2.18.1.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for a discussion of this feature.

**buffer**(*Buffering*)

Defines output buffering. The atom full (default) defines full buffering, line buffering by line, and false implies the stream is fully unbuffered. Smaller buffering is useful if another process or the user is waiting for the output as it is being produced. See alsoflush\_output/[0,1]. This option is not an ISO option.

**close\_on\_abort**(*Bool*)

If true (default), the stream is closed on an abort (see [abort/0](http://www.swi-prolog.org/pldoc/man?predicate=abort/0)). If false, the stream is not closed. If it is an output stream, however, it will be flushed. Useful for logfiles and if the stream is associated to a process (using the pipe/1 construct).

**create**(*+List*)

Specifies how a new file is created when opening in write, append or update mode. Currently, *List* is a list of atoms that describe the permissions of the created file.73Defined values are below. Not recognised values are silently ignored, allowing for adding platform specific extensions to this set.

**read**

Allow read access to the file.

**write**

Allow write access to the file.

**execute**

Allow execution access to the file.

**default**

Allow read and write access to the file.

**all**

Allow any access provided by the OS.

Note that if *List* is empty, the created file has no associated access permissions. The create options map to the POSIX *mode* option of open(), where read map to 0444, writeto 0222 and execute to 0111. On POSIX systems, the final permission is defined as (mode & ~umask).

**encoding**(*Encoding*)

Define the encoding used for reading and writing text to this stream. The default encoding for type text is derived from the Prolog flag [encoding](http://www.swi-prolog.org/pldoc/man?section=flags#flag:encoding). For binary streams the default encoding is octet. For details on encoding issues, see [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars).

**eof\_action**(*Action*)

Defines what happens if the end of the input stream is reached. Action eof\_code makes[get0/1](http://www.swi-prolog.org/pldoc/man?predicate=get0/1) and friends return -1, and [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) and friends return the atom end\_of\_file. Repetitive reading keeps yielding the same result. Action error is like eof\_code, but repetitive reading will raise an error. With action reset, Prolog will examine the file again and return more data if the file has grown.

**locale**(*+Locale*)

Set the locale that is used by notably [format/2](http://www.swi-prolog.org/pldoc/man?predicate=format/2) for output on this stream. See [section 4.22](http://www.swi-prolog.org/pldoc/man?section=locale).

**lock**(*LockingMode*)

Try to obtain a lock on the open file. Default is none, which does not lock the file. The value read or shared means other processes may read the file, but not write it. The value write or exclusive means no other process may read or write the file.

Locks are acquired through the POSIX function fcntl() using the command F\_SETLKW, which makes a blocked call wait for the lock to be released. Please note that fcntl() locks are *advisory* and therefore only other applications using the same advisory locks honour your lock. As there are many issues around locking in Unix, especially related to NFS (network file system), please study the fcntl() manual page before trusting your locks!

The lock option is a SWI-Prolog extension.

**type**(*Type*)

Using type text (default), Prolog will write a text file in an operating system compatible way. Using type binary the bytes will be read or written without any translation. See also the option encoding.

**wait**(*Bool*)

This option can be combined with the lock option. If false (default true), the open call returns immediately with an exception if the file is locked. The exception has the formatpermission\_error(lock, source\_sink, SrcDest).

The option reposition is not supported in SWI-Prolog. All streams connected to a file may be repositioned.

**open\_dde\_conversation**(*+Service, +Topic, -Handle*)

Open a conversation with a server supporting the given service name and topic (atoms). If successful, *Handle* may be used to send transactions to the server. If no willing server is found this predicate fails silently.

**open\_null\_stream**(*--Stream*)

Open an output stream that produces no output. All counting functions are enabled on such a stream. It can be used to discard output (like Unix /dev/null) or exploit the counting properties. The initial encoding of *Stream* is utf8, enabling arbitrary Unicode output. The encoding can be changed to determine byte counts of the output in a particular encoding or validate if output is possible in a particular encoding. For example, the code below determines the number of characters emitted when writing *Term*.

write\_length(Term, Len) :-

open\_null\_stream(Out),

write(Out, Term),

character\_count(Out, Len0),

close(Out),

Len = Len0.

**open\_resource**(*+Name, ?Class, -Stream*)

Opens the resource specified by *Name* and *Class*. If the latter is a variable, it will be unified to the class of the first resource found that has the specified *Name*. If successful, *Stream*becomes a handle to a binary input stream, providing access to the content of the resource.

The predicate [open\_resource/3](http://www.swi-prolog.org/pldoc/man?predicate=open_resource/3) first checks [resource/3](http://www.swi-prolog.org/pldoc/man?predicate=resource/3). When successful it will open the returned resource source file. Otherwise it will look in the program's resource database. When creating a saved state, the system normally saves the resource contents into the resource archive, but does not save the resource clauses.

This way, the development environment uses the files (and modifications) to the [resource/3](http://www.swi-prolog.org/pldoc/man?predicate=resource/3)declarations and/or files containing resource info, thus immediately affecting the running environment, while the runtime system quickly accesses the system resources.

**open\_shared\_object**(*+File, -Handle*)

*File* is the name of a shared object file (DLL in MS-Windows). This file is attached to the current process, and *Handle* is unified with a handle to the library. Equivalent toopen\_shared\_object(File, Handle, []). See also [open\_shared\_object/3](http://www.swi-prolog.org/pldoc/man?predicate=open_shared_object/3) and[load\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=load_foreign_library/1).

On errors, an exception shared\_object(Action, Message) is raised. *Message* is the return value from dlerror().

**open\_shared\_object**(*+File, -Handle, +Options*)

As [open\_shared\_object/2](http://www.swi-prolog.org/pldoc/man?predicate=open_shared_object/2), but allows for additional flags to be passed. *Options* is a list of atoms. now implies the symbols are resolved immediately rather than lazy (default). globalimplies symbols of the loaded object are visible while loading other shared objects (by default they are local). Note that these flags may not be supported by your operating system. Check the documentation of dlopen() or equivalent on your operating system. Unsupported flags are silently ignored.

**pen\_string**(*+String, -Stream*)

True when *Stream* is an input stream that accesses the content of *String*. *String* can be any text representation, i.e., string, atom, list of codes or list of characters.

**ord\_list\_to\_assoc**(*+List, -Assoc*)

*Assoc* is an association list correspond to the Key-Value pairs in *List*, which must occur in strictly ascending order of their keys.

**parse\_time**(*+Text, -Stamp*)

Same as parse\_time(Text, \_Format, Stamp). See [parse\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=parse_time/3).

**parse\_time**(*+Text, ?Format, -Stamp*)

Parse a textual time representation, producing a time-stamp. Supported formats for *Text* are in the table below. If the format is known, it may be given to reduce parse time and avoid ambiguities. Otherwise, *Format* is unified with the format encountered.

|  |  |
| --- | --- |
| **Name** | **Example** |
| rfc\_1123 | Fri, 08 Dec 2006 15:29:44 GMT |
|  | Fri, 08 Dec 2006 15:29:44 +0000 |
| iso\_8601 | 2006-12-08T17:29:44+02:00 |
|  | 20061208T172944+0200 |
|  | 2006-12-08T15:29Z |
|  | 2006-12-08 |
|  | 20061208 |
|  | 2006-12 |
|  | 2006-W49-5 |
|  | 2006-342 |

**pce\_dispatch**(*+Options*)

Create a Prolog thread with the alias name pce for XPCE event handling. In the X11 version this call creates a thread that executes the X11 event-dispatch loop. In MS-Windows it creates a thread that executes a windows event-dispatch loop. The XPCE event-handling thread has the alias pce. *Options* specifies the thread attributes as [thread\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_create/3).

**pce\_call***(:Goal)* is **det**

Run *Goal* in the XPCE thread.

**deprecated**

- New code should used [in\_pce\_thread/1](http://www.swi-prolog.org/pldoc/man?predicate=in_pce_thread/1).

**peek\_byte**(*-Byte*)

*[ISO]***peek\_byte**(*+Stream, -Byte*)

*[ISO]***peek\_code**(*-Code*)

*[ISO]***peek\_code**(*+Stream, -Code*)

*[ISO]***peek\_char**(*-Char*)

*[ISO]***peek\_char**(*+Stream, -Char*)

Read the next byte/code/char from the input without removing it. These predicates do not modify the stream's position or end-of-file status. These predicates require a buffered stream (see [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2)) and raise a permission error if the stream is unbuffered or the buffer is too small to hold the longest multi-byte sequence that might need to be buffered.

|  |  |
| --- | --- |
| [**peek\_byte/2**](http://www.swi-prolog.org/pldoc/chario.html#peek_byte/2) | Read byte without removing |
| [**peek\_char/1**](http://www.swi-prolog.org/pldoc/chario.html#peek_char/1) | Read character without removing |
| [**peek\_char/2**](http://www.swi-prolog.org/pldoc/chario.html#peek_char/2) | Read character without removing |
| [**peek\_code/1**](http://www.swi-prolog.org/pldoc/chario.html#peek_code/1) | Read character-code without removing |
| [**peek\_code/2**](http://www.swi-prolog.org/pldoc/chario.html#peek_code/2) | Read character-code without removing |

**peek\_string**(*+Stream, +Len, -String*)

Read the next *Len* characters (if the stream is a text stream) or bytes (if the stream is binary) from Stream without removing the data. If *Len* is larger that the stream buffer size, the buffer size is increased to *Len*. *String* can be shorter than *Len* if the stream contains less data. This predicate is intended to guess the content type of data read from non-repositionable streams.

**phrase**(*:DCGBody, ?List*)

Equivalent to phrase(*DCGBody*, *InputList*, []).

**phrase**(*:DCGBody, ?List, ?Rest*)

True when *DCGBody* applies to the difference *List*/*Rest*. Although *DCGBody* is typically a*callable* term that denotes a grammar rule, it can be any term that is valid as the body of a DCG rule.

The example below calls the rule set integer/3 defined in [section 4.12](http://www.swi-prolog.org/pldoc/man?section=DCG) and available fromlibrary(library(dcg/basics)), binding *Rest* to the remainder of the input after matching the integer.

?- [library(dcg/basics)].

?- atom\_codes('42 times', Codes),

phrase(integer(X), Codes, Rest).

X = 42

Rest = [32, 116, 105, 109, 101, 115]

The next example exploits a complete body. Given the following definition of digit\_weight//1 , we can pose the query below.

digit\_weight(W) -->

[D],

{ code\_type(D, digit(W)) }.

?- atom\_codes('Version 3.4', Codes),

phrase(("Version ",

digit\_weight(Major),".",digit\_weight(Minor)),

Codes).

Major = 3,

Minor = 4.

The SWI-Prolog implementation of [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3) verifies that the *List* and *Rest* arguments are unbound, bound to the empty list or a list *cons cell*. Other values raise a type error.61 The predicate [call\_dcg/3](http://www.swi-prolog.org/pldoc/man?predicate=call_dcg/3) is provided to use grammar rules with terms that are not lists.

Note that the syntax for lists of codes changed in SWI-Prolog version 7 (see [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)). If a DCG body is translated, both "text" and `text` is a valid code-list literal in version 7. A version 7 string ("text") is **not** acceptable for the second and third arguments of [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3). This is typically not a problem for applications as the input of a DCG rarely appears in the source code. For testing in the toplevel, one must use double quoted text in versions prior to 7 and back quoted text in version 7 or later.

See also [portray\_text/1](http://www.swi-prolog.org/pldoc/man?predicate=portray_text/1), which can be used to print lists of character codes as a string to the top level and debugger to facilitate debugging DCGs that process character codes. The librarylibrary(apply\_macros) compiles [phrase/3](http://www.swi-prolog.org/pldoc/man?predicate=phrase/3) if the argument is sufficiently instantiated, eliminating the runtime overhead of translating *DCGBody* and meta-calling.

**phrase\_from\_quasi\_quotation**(*:Grammar, +Content*)

Process the quasi quotation using the DCG *Grammar*. Failure of the grammer is interpreted as a syntax error.

**See also**

[with\_quasi\_quotation\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=with_quasi_quotation_input/3) for processing quotations from stream.

|  |  |
| --- | --- |
| **please/3** | Query/change environment parameters |

**plus**(*?Int1, ?Int2, ?Int3*)

True if *Int3 = Int1 + Int2*. At least two of the three arguments must be instantiated to integers.

**portray**(*+Term*)

A dynamic predicate, which can be defined by the user to change the behaviour of [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) on (sub)terms. For each subterm encountered that is not a variable [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) first calls [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1)using the term as argument. For lists, only the list as a whole is given to [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1). If [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1)succeeds [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) assumes the term has been written.

**portray\_clause**(*+Clause*)

Pretty print a clause. A clause should be specified as a term `<*Head*> :- <*Body*>'. Facts are represented as `<*Head*> :- true' or simply <*Head*>. Variables in the clause are written as *A*, *B*, ... . Singleton variables are written as \_. See also [portray\_clause/2](http://www.swi-prolog.org/pldoc/man?predicate=portray_clause/2).

**portray\_clause**(*+Stream, +Clause*)

Pretty print a clause to *Stream*. See [portray\_clause/1](http://www.swi-prolog.org/pldoc/man?predicate=portray_clause/1) for details.

**predicate\_property**(*:Head, ?Property*)

True when *Head* refers to a predicate that has property *Property*. With sufficiently instantiated*Head*, [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) tries to resolve the predicate the same way as calling it would do: if the predicate is not defined it scans the default modules (see [default\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=default_module/2)) and finally tries the autoloader. Unlike calling, failure to find the target predicate causes[predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2) to fail silently. If *Head* is not sufficiently bound, only currently locally defined and already imported predicates are enumerated. See [current\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=current_predicate/1) for enumerating all predicates. A common issue concerns *generating* all built-in predicates. This can be achieved using the code below:

generate\_built\_in(Name/Arity) :-

predicate\_property(system:Head, built\_in),

functor(Head, Name, Arity),

\+ sub\_atom(Name, 0, \_, \_, $). % discard reserved names

*Property* is one of:

**autoload**(*File*)

True if the predicate can be autoloaded from the file *File*. Like undefined, this property is *not* generated.

**built\_in**

True if the predicate is locked as a built-in predicate. This implies it cannot be redefined in its definition module and it can normally not be seen in the tracer.

**defined**

True if the predicate is defined. This property is aware of sources being *reloaded*, in which case it claims the predicate defined only if it is defined in another source or it has seen a definition in the current source. See [compile\_aux\_clauses/1](http://www.swi-prolog.org/pldoc/man?predicate=compile_aux_clauses/1).

**dynamic**

True if [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1) and [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1) may be used to modify the predicate. This property is set using [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1).

**exported**

True if the predicate is in the public list of the context module.

**imported\_from**(*Module*)

Is true if the predicate is imported into the context module from module *Module*.

**file**(*FileName*)

Unify *FileName* with the name of the source file in which the predicate is defined. See also [source\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file/2) and the property line\_count. Note that this reports the file of the first clause of a predicate. A more robust interface can be achieved using [nth\_clause/3](http://www.swi-prolog.org/pldoc/man?predicate=nth_clause/3)and [clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2).

**foreign**

True if the predicate is defined in the C language.

**indexed**(*Indexes*)

*Indexes*70 is a list of additional (hash) indexes on the predicate. Each element of the list is a term *ArgSpec*-*Index*. Currently *ArgSpec* is an integer denoting the argument position and *Index* is a term hash(Buckets, Speedup, IsList). Here *Buckets* is the number of buckets in the hash and *Speedup* is the expected speedup relative to trying all clauses linearly. *IsList* indicates that a list is created for all clauses with the same key. This is currently not used.

**interpreted**

True if the predicate is defined in Prolog. We return true on this because, although the code is actually compiled, it is completely transparent, just like interpreted code.

**iso**

True if the predicate is covered by the ISO standard (ISO/IEC 13211-1).

**line\_count**(*LineNumber*)

Unify *LineNumber* with the line number of the first clause of the predicate. Fails if the predicate is not associated with a file. See also [source\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file/2). See also the file property above, notably the reference to [clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2).

**multifile**

True if there may be multiple (or no) files providing clauses for the predicate. This property is set using [multifile/1](http://www.swi-prolog.org/pldoc/man?predicate=multifile/1).

**meta\_predicate**(*Head*)

If the predicate is declared as a meta-predicate using [meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1), unify *Head* with the head-pattern. The head-pattern is a compound term with the same name and arity as the predicate where each argument of the term is a meta-predicate specifier. See[meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1) for details.

**nodebug**

Details of the predicate are not shown by the debugger. This is the default for built-in predicates. User predicates can be compiled this way using the Prolog flag[generate\_debug\_info](http://www.swi-prolog.org/pldoc/man?section=flags#flag:generate_debug_info).

**notrace**

Do not show ports of this predicate in the debugger.

**number\_of\_clauses**(*ClauseCount*)

Unify *ClauseCount* to the number of clauses associated with the predicate. Fails for foreign predicates.

**number\_of\_rules**(*RuleCount*)

Unify *RuleCount* to the number of clauses associated with the predicate. A *rule* is defined as a clauses that has a body that is not just true (i.e., a *fact*). Fails for foreign predicates. This property is used to avoid analysing predicates with only facts inlibrary(prolog\_codewalk).

**public**

Predicate is declared public using [public/1](http://www.swi-prolog.org/pldoc/man?predicate=public/1). Note that without further definition, public predicates are considered undefined and this property is *not* reported.

**quasi\_quotation\_syntax**

The predicate (with arity 4) is declared to provide quasi quotation syntax with[quasi\_quotation\_syntax/1](http://www.swi-prolog.org/pldoc/man?predicate=quasi_quotation_syntax/1).

**static**

The definition can *not* be modified using [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1) and friends. This property is the opposite from dynamic, i.e., for each defined predicate, either static or dynamic is true but never both.

**thread\_local**

If true (only possible on the multithreaded version) each thread has its own clauses for the predicate. This property is set using [thread\_local/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_local/1).

**transparent**

True if the predicate is declared transparent using the [module\_transparent/1](http://www.swi-prolog.org/pldoc/man?predicate=module_transparent/1) or[meta\_predicate/1](http://www.swi-prolog.org/pldoc/man?predicate=meta_predicate/1) declaration. In the latter case the property meta\_predicate(Head) is also provided. See [chapter 6](http://www.swi-prolog.org/pldoc/man?section=modules) for details.

**undefined**

True if a procedure definition block for the predicate exists, but there are no clauses for it and it is not declared dynamic or multifile. This is true if the predicate occurs in the body of a loaded predicate, an attempt to call it has been made via one of the meta-call predicates, the predicate has been declared as e.g., a meta-predicate or the predicate had a definition in the past. See the library package library(check) for example usage.

**visible**

True when predicate can be called without raising a predicate existence error. This means that the predicate is (1) defined, (2) can be inherited from one of the default modules (see [default\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=default_module/2)) or (3) can be autoloaded. The behaviour is logically consistent iff the property visible is provided explicitly. If the property is left unbound, only defined predicates are enumerated.

**volatile**

If true, the clauses are not saved into a saved state by qsave\_program/[1,2]. This property is set using [volatile/1](http://www.swi-prolog.org/pldoc/man?predicate=volatile/1).

**predsort**(*+Pred, +List, -Sorted*)

Sorts similar to [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2), but determines the order of two terms by calling *Pred*(-*Delta*, +*E1*, +*E2*) . This call must unify *Delta* with one of <, > or =. If the built-in predicate [compare/3](http://www.swi-prolog.org/pldoc/man?predicate=compare/3) is used, the result is the same as [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2). See also [keysort/2](http://www.swi-prolog.org/pldoc/man?predicate=keysort/2).

**print**(*+Term*)

Print a term for debugging purposes. The predicate [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) acts as if defined as below.

print(Term) :-

current\_prolog\_flag(print\_write\_options, Options), !,

write\_term(Term, Options).

print(Term) :-

write\_term(Term, [ portray(true),

numbervars(true),

quoted(true)

]).

The [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) predicate is used primarily through the ~p escape sequence of [format/2](http://www.swi-prolog.org/pldoc/man?predicate=format/2), which is commonly used in the recipies used by [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) to emit messages.

The classical definition of this predicate is equivalent to the ISO predicate [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) using the options portray(true) and numbervars(true). The portray(true) option allows the user to implement application-specific printing of terms printed during debugging to facilitate easy understanding of the output. See also [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) and library(portray\_text). SWI-Prolog adds quoted(true) to (1) facilitate the copying/pasting of terms that are not affected by [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) and to (2) allow numbers, atoms and strings to be more easily distinguished, e.g.,42, '42' and "42".

**print**(*+Stream, +Term*)

Print *Term* to *Stream*.

**print\_message**(*+Kind, +Term*)

The predicate [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) is used by the system and libraries to print messages. *Kind*describes the nature of the message, while *Term* is a Prolog term that describes the content. Printing messages through this indirection instead of using [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3) to the streamuser\_error allows displaying the message appropriate to the application (terminal, logfile, graphics), acting on messages based on their content instead of a string (see[message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3)) and creating language specific versions of the messages. See also [section 4.10.3.1](http://www.swi-prolog.org/pldoc/man?section=exception). The following message kinds are known:

**banner**

The system banner message. Banner messages can be suppressed by setting the Prolog flag [verbose](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose) to silent.

**debug**(*Topic*)

Message from library(debug). See [debug/3](http://www.swi-prolog.org/pldoc/man?predicate=debug/3).

**error**

The message indicates an erroneous situation. This kind is used to print uncaught exceptions of type error(Formal, Context). See section introduction ([section 4.10.3](http://www.swi-prolog.org/pldoc/man?section=exception)).

**help**

User requested help message, for example after entering `h' or `?' to a prompt.

**information**

Information that is requested by the user. An example is [statistics/0](http://www.swi-prolog.org/pldoc/man?predicate=statistics/0).

**informational**

Typically messages of events are progres that are considered useful to a developer. Such messages can be suppressed by setting the Prolog flag [verbose](http://www.swi-prolog.org/pldoc/man?section=flags#flag:verbose) to silent.

**silent**

Message that is normally not printed. Applications may define [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3) to act upon such messages.

**trace**

Messages from the (command line) tracer.

**warning**

The message indicates something dubious that is not considered fatal. For example, discontiguous predicates (see [discontiguous/1](http://www.swi-prolog.org/pldoc/man?predicate=discontiguous/1)).

The predicate [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) first translates the *Term* into a list of `message lines' (see[print\_message\_lines/3](http://www.swi-prolog.org/pldoc/man?predicate=print_message_lines/3) for details). Next, it calls the hook [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3) to allow the user to intercept the message. If [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3) fails it prints the message unless *Kind* is silent.

The [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) predicate and its rules are in the file <*plhome*>/boot/messages.pl, which may be inspected for more information on the error messages and related error terms. If you need to write messages from your own predicates, it is recommended to reuse the existing message terms if applicable. If no existing message term is applicable, invent a fairly unique term that represents the event and define a rule for the multifile predicate prolog:message//1. See [section 4.10.3.1](http://www.swi-prolog.org/pldoc/man?section=exception) for a deeper discussion and examples.

See also [message\_to\_string/2](http://www.swi-prolog.org/pldoc/man?predicate=message_to_string/2).

**print\_message\_lines**(*+Stream, +Prefix, +Lines*)

Print a message (see [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2)) that has been translated to a list of message elements. The elements of this list are:

**<*Format*>-<*Args*>**

Where *Format* is an atom and *Args* is a list of format arguments. Handed to [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3).

**flush**

If this appears as the last element, *Stream* is flushed (see [flush\_output/1](http://www.swi-prolog.org/pldoc/man?predicate=flush_output/1)) and no final newline is generated. This is combined with a subsequent message that starts withat\_same\_line to complete the line.

**at\_same\_line**

If this appears as first element, no prefix is printed for the first line and the line position is not forced to 0 (see [format/1](http://www.swi-prolog.org/pldoc/man?predicate=format/1), ~N).

**ansi**(*+Attributes, +Format, +Args*)

This message may be intercepted by means of the hook prolog:message\_line\_element/2. The library library(ansi\_term) implements this hook to achieve coloured output. If it is not intercepted it invokes format(Stream, Format, Args).

**nl**

A new line is started. If the message is not complete, *Prefix* is printed before the remainder of the message.

**begin**(*Kind, Var*)

**end**(*Var*)

The entire message is headed by begin(Kind, Var) and ended by end(Var). This feature is used by, e.g., library library(ansi\_term) to colour entire messages.

**<*Format*>**

Handed to [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3) as format(Stream, Format,[]). Deprecated because it is ambiguous if *Format* collides with one of the atomic commands.

See also [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) and [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3).

**profile**(*:Goal*)

Execute *Goal* just like [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1), collecting profiling statistics, and call show\_profile([]). With XPCE installed this opens a graphical interface to examine the collected profiling data.

**profile***(:Goal)*

**profile***(:Goal, +Options)*

Run *Goal* under the execution profiler. Defined options are:

**time***(Which)*

Profile cpu or wall time. The default is CPU time.

**top***(N)*

When generating a textual report, show the top *N* predicates.

**cumulative***(Bool)*

If true (default false), show cumulative output in a textual report.

|  |  |
| --- | --- |
| **profile\_count/3** | Obtain profile results on a predicate |
|  |  |

**profiler**(*-Old, +New*)

Query or change the status of the profiler. The status is one of

**false**

The profiler is not activated.

**cputime**

The profiler collects CPU statistics.

**walltime**

The profiler collects wall time statistics.

The value true is accepted as a synonym for cputime for compatibility reasons.

**prolog**

This goal starts the default interactive top level. Queries are read from the stream user\_input. See also the Prolog flag [history](http://www.swi-prolog.org/pldoc/man?section=flags#flag:history). The [prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0) predicate is terminated (succeeds) by typing the end-of-file character (typically control-D).

**prolog\_choice\_attribute**(*+ChoicePoint, +Key, -Value*)

Extract attributes of a choice point. *ChoicePoint* is a reference to a choice point as passed to[prolog\_trace\_interception/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_trace_interception/4) on the 3rd argument or obtained using[prolog\_current\_choice/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_current_choice/1). *Key* specifies the requested information:

**parent**

Requests a reference to the first older choice point.

**frame**

Requests a reference to the frame to which the choice point refers.

**type**

Requests the type. Defined values are clause (the goal has alternative clauses), foreign(non-deterministic foreign predicate), jump (clause internal choice point), top (first dummy choice point), catch ([catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) to allow for undo), debug (help the debugger), ornone (has been deleted).

This predicate is used for the graphical debugger to show the choice point stack.

**prolog\_current\_choice**(*-Choice*)

Unify *Choice* with an integer provided a reference to the last choice point. Fails if the current environment has no choice points. See also [prolog\_choice\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_choice_attribute/3).

**prolog\_current\_frame**(*-Frame*)

Unify *Frame* with an integer providing a reference to the parent of the current local stack frame. A pointer to the current local frame cannot be provided as the predicate succeeds deterministically and therefore its frame is destroyed immediately after succeeding.

**prolog\_cut\_to**(*+Choice*)

Prunes all choice points created since *Choice*. Can be used together with[prolog\_current\_choice/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_current_choice/1) to implement *ancestral* cuts. This predicate is in the hackers corner because it should not be used in normal Prolog code. It may be used to create new high level control structures, particularly for compatibility purposes.

Note that in the current implementation, the pruned choice points and environment frames are *not* reclaimed. As a consequence, where predicates that are deterministic due to clause indexing, normal cuts or (if\send{then};else) and and tail recursive run in bounded local stack space, predicates using [prolog\_cut\_to/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_cut_to/1) will run out of stack.

**locate***(+Spec, -Location)[multifile,* [*library(edit)*](http://www.swi-prolog.org/pldoc/doc/swi/library/edit.pl)*]*

Locate object from the specified location.

**edit\_command***(+Editor, -Command)[multifile,* [*library(edit)*](http://www.swi-prolog.org/pldoc/doc/swi/library/edit.pl)*]*

This predicate should specify the shell-command called to invoke the user's editor. The following substitutions will be made:

|  |  |
| --- | --- |
| %e | Path name of the editor |
| %f | Path name of the file to be edited |
| %d | Line number of the target |

**prolog\_exception\_hook**(*+ExceptionIn, -ExceptionOut, +Frame, +CatcherFrame*)

This hook predicate, if defined in the module user, is between raising an exception and handling it. It is intended to allow a program adding additional context to an exception to simplify diagnosing the problem. *ExceptionIn* is the exception term as raised by [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1) or one of the built-in predicates. The output argument *ExceptionOut* describes the exception that is actually raised. *Frame* is the innermost frame. See [prolog\_frame\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_frame_attribute/3) and the library library(prolog\_stack) for getting information from this. *CatcherFrame* is a reference to the frame calling the matching [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3), none if the exception is not caught or 'C'if the exception is caught in C calling Prolog using the flag PL\_Q\_CATCH\_EXCEPTION.

The hook is run in `nodebug' mode. If it succeeds, *ExceptionOut* is considered the current exception. If it fails, *ExceptionIn* is used for further processing. The hook is *never* called recursively. The hook is *not* allowed to modify *ExceptionOut* in such a way that it no longer unifies with the catching frame.

Typically, [prolog\_exception\_hook/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_exception_hook/4) is used to fill the second argument of error(Formal, Context) exceptions. *Formal* is defined by the ISO standard, while SWI-Prolog defines *Context*as a term context(Location, Message). *Location* is bound to a term <*name*>/<*arity*> by the kernel. This hook can be used to add more information on the calling context, such as a full stack trace.

Applications that use exceptions as part of normal processing must do a quick test of the environment before starting expensive gathering information on the state of the program.

The hook can call [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) to enter trace mode immediately. For example, imagine an application performing an unwanted division by zero while all other errors are expected and handled. We can force the debugger using the hook definition below. Run the program in debug mode (see [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0)) to preserve as much as possible of the error context.

user:prolog\_exception\_hook(

error(evaluation\_error(zero\_divisor), \_),

\_, \_, \_) :-

trace, fail.

**prolog\_file\_type**(*?Extension, ?Type*)

This dynamic multifile predicate defined in module user determines the extensions considered by [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2). *Extension* is the filename extension without the leading dot, and *Type* denotes the type as used by the file\_type(Type) option of [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2). Here is the initial definition of [prolog\_file\_type/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_file_type/2):

user:prolog\_file\_type(pl, prolog).

user:prolog\_file\_type(Ext, prolog) :-

current\_prolog\_flag(associate, Ext),

Ext \== pl.

user:prolog\_file\_type(qlf, qlf).

user:prolog\_file\_type(Ext, executable) :-

current\_prolog\_flag(shared\_object\_extension, Ext).

Users can add extensions for Prolog source files to avoid conflicts (for example with **perl**) as well as to be compatible with another Prolog implementation. We suggest using .pro for avoiding conflicts with **perl**. Overriding the system definitions can stop the system from finding libraries.

**prolog\_frame\_attribute**(*+Frame, +Key, :Value*)

Obtain information about the local stack frame *Frame*. *Frame* is a frame reference as obtained through [prolog\_current\_frame/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_current_frame/1), [prolog\_trace\_interception/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_trace_interception/4) or this predicate. The key values are described below.

**alternative**

*Value* is unified with an integer reference to the local stack frame in which execution is resumed if the goal associated with *Frame* fails. Fails if the frame has no alternative frame.

**has\_alternatives**

*Value* is unified with true if *Frame* still is a candidate for backtracking; false otherwise.

**goal**

*Value* is unified with the goal associated with *Frame*. If the definition module of the active predicate is not the calling context, the goal is represented as <*module*>:<*goal*>. Do not instantiate variables in this goal unless you **know** what you are doing! Note that the returned term may contain references to the frame and should be discarded before the frame terminates.167

**parent\_goal**

If *Value* is instantiated to a callable term, find a frame executing the predicate described by *Value* and unify the arguments of *Value* to the goal arguments associated with the frame. This is intended to check the current execution context. The user must ensure the checked parent goal is not removed from the stack due to last-call optimisation and be aware of the slow operation on deeply nested calls.

**predicate\_indicator**

Similar to goal, but only returning the [<*module*>:]<*name*>/<*arity*> term describing the term, not the actual arguments. It avoids creating an illegal term as goal and is used by the library library(prolog\_stack).

**clause**

*Value* is unified with a reference to the currently running clause. Fails if the current goal is associated with a foreign (C) defined predicate. See also [nth\_clause/3](http://www.swi-prolog.org/pldoc/man?predicate=nth_clause/3) and[clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2).

**level**

*Value* is unified with the recursion level of *Frame*. The top level frame is at level `0'.

**parent**

*Value* is unified with an integer reference to the parent local stack frame of *Frame*. Fails if *Frame* is the top frame.

**context\_module**

*Value* is unified with the name of the context module of the environment.

**top**

*Value* is unified with true if *Frame* is the top Prolog goal from a recursive call back from the foreign language; false otherwise.

**hidden**

*Value* is unified with true if the frame is hidden from the user, either because a parent has the hide-childs attribute (all system predicates), or the system has no trace-me attribute.

**skipped**

*Value* is true if this frame was skipped in the debugger.

**pc**

*Value* is unified with the program pointer saved on behalf of the parent goal if the parent goal is not owned by a foreign predicate or belongs to a compound meta-call (e.g., call((a,b))).

**argument**(*N*)

*Value* is unified with the *N*-th slot of the frame. Argument 1 is the first argument of the goal. Arguments above the arity refer to local variables. Fails silently if *N* is out of range.

**prolog\_ide**(*+Action*)

This predicate ensures the IDE-enabling XPCE component is loaded, creates the XPCE class**prolog\_ide** and sends *Action* to its one and only instance @prolog\_ide. *Action* is one of the following:

**open\_navigator**(*+Directory*)

Open the Prolog Navigator (see [section 3.6](http://www.swi-prolog.org/pldoc/man?section=navigator)) in the given *Directory*.

**open\_debug\_status**

Open a window to edit spy and trace points.

**open\_query\_window**

Open a little window to run Prolog queries from a GUI component.

**thread\_monitor**

Open a graphical window indicating existing threads and their status.

**debug\_monitor**

Open a graphical front-end for the library(debug) library that provides an overview of the topics and catches messages.

**xref**

Open a graphical front-end for the cross-referencer that provides an overview of predicates and their callers.

**prolog\_list\_goal**(*:Goal*)

Hook, normally not defined. This hook is called by the 'L' command of the tracer in the module user to list the currently called predicate. This hook may be defined to list only relevant clauses of the indicated *Goal* and/or show the actual source code in an editor. See also [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) and [multifile/1](http://www.swi-prolog.org/pldoc/man?predicate=multifile/1).

**prolog\_load\_context**(*?Key, ?Value*)

Obtain context information during compilation. This predicate can be used from directives appearing in a source file to get information about the file being loaded as well as by the[term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2) and [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) hooks. See also [source\_location/2](http://www.swi-prolog.org/pldoc/man?predicate=source_location/2) and [if/1](http://www.swi-prolog.org/pldoc/man?predicate=if/1). The following keys are defined:

|  |  |
| --- | --- |
| **Key** | **Description** |
| directory | Directory in which source lives |
| dialect | Compatibility mode. See [expects\_dialect/1](http://www.swi-prolog.org/pldoc/man?predicate=expects_dialect/1). |
| file | Similar to source, but returns the file being included when called while an include file is being processed |
| module | Module into which file is loaded |
| reload | true if the file is being **re**loaded. Not present on first load |
| script | Boolean that indicates whether the file is loaded as a script file (see**-s**) |
| source | File being loaded. If the system is processing an included file, the value is the *main* file. Returns the original Prolog file when loading a .qlf file. |
| stream | Stream identifier (see [current\_input/1](http://www.swi-prolog.org/pldoc/man?predicate=current_input/1)) |
| term\_position | Start position of last term read. See also [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2)(position property and [stream\_position\_data/3](http://www.swi-prolog.org/pldoc/man?predicate=stream_position_data/3).43 |
| term | Term being expanded by [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2). |
| variable\_names | A list of `*Name* = *Var*' of the last term read. See [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) for details. |

The directory is commonly used to add rules to [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2), setting up a search path for finding files with [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3). For example:

:- dynamic user:file\_search\_path/2.

:- multifile user:file\_search\_path/2.

:- prolog\_load\_context(directory, Dir),

asserta(user:file\_search\_path(my\_program\_home, Dir)).

...

absolute\_file\_name(my\_program\_home('README.TXT'), ReadMe,

[ access(read) ]),

...

**prolog\_load\_file**(*+Spec, +Options*)

Load a single object. If this call succeeds, [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) assumes the action has been taken care of. This hook is only called if *Options* does not contain the stream(Input) option. The hook must be defined in the module user.

This can be used to load from unusual places. For example, librarylibrary(http/http\_load) loads Prolog directly from an HTTP server. It can also be used to load source in unusual forms, such as loading compressed files without decompressing them first. There is currently no example of that.

**prolog\_skip\_level**(*-Old, +New*)

Unify *Old* with the old value of `skip level' and then set this level according to *New*. *New* is an integer, the atom very\_deep (meaning don't skip) or the atom skip\_in\_redo (see[prolog\_skip\_frame/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_skip_frame/1)). The `skip level' is a setting of each Prolog thread that disables the debugger on all recursion levels deeper than the level of the variable. See also[prolog\_skip\_frame/1](http://www.swi-prolog.org/pldoc/man?predicate=prolog_skip_frame/1).

**prolog\_skip\_frame**(*-Frame*)

Indicate *Frame* as a skipped frame and set the `skip level' (see [prolog\_skip\_level/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_skip_level/2) to the recursion depth of *Frame*. The effect of the skipped flag is that a redo on a child of this frame is handled differently. First, a redo trace is called for the child, where the skip level is set toredo\_in\_skip. Next, the skip level is set to skip level of the skipped frame.

**prolog\_stack\_property**(*?Stack, ?KeyValue*)

True if *KeyValue* is a current property of *Stack*. See [set\_prolog\_stack/2](http://www.swi-prolog.org/pldoc/man?predicate=set_prolog_stack/2) for defined properties.

**prolog\_to\_os\_filename**(*?PrologPath, ?OsPath*)

Convert between the internal Prolog path name conventions and the operating system path name conventions. The internal conventions follow the POSIX standard, which implies that this predicate is equivalent to =/2 (unify) on POSIX (e.g., Unix) systems. On Windows systems it changes the directory separator from \ into /.

**prolog\_trace\_interception**(*+Port, +Frame, +Choice, -Action*)

Dynamic predicate, normally not defined. This predicate is called from the SWI-Prolog debugger just before it would show a port. If this predicate succeeds, the debugger assumes that the trace action has been taken care of and continues execution as described by *Action*. Otherwise the normal Prolog debugger actions are performed.

*Port* denotes the reason to activate the tracer (`port' in the 4/5-port, but with some additions):

**call**

Normal entry through the call port of the 4-port debugger.

**redo**(*PC*)

Normal entry through the redo port of the 4-port debugger. The redo port signals resuming a predicate to generate alternative solutions. If *PC* is 0 (zero), clause indexing has found another clause that will be tried next. Otherwise, *PC* is the program counter in the current clause where execution continues. This implies we are dealing with an in-clause choice point left by, e.g., [;/2](http://www.swi-prolog.org/pldoc/man?predicate=%3B/2). Note that non-determinism in foreign predicates are also handled using an in-clause choice point.

**unify**

The unify port represents the *neck* instruction, signalling the end of the head-matching process. This port is normally invisible. See [leash/1](http://www.swi-prolog.org/pldoc/man?predicate=leash/1) and [visible/1](http://www.swi-prolog.org/pldoc/man?predicate=visible/1).

**exit**

The exit port signals the goal is proved. It is possible for the goal to have alternatives. See [prolog\_frame\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_frame_attribute/3) to examine the goal stack.

**fail**

The fail port signals final failure of the goal.

**exception**(*Except*)

An exception is raised and still pending. This port is activated on each parent frame of the frame generating the exception until the exception is caught or the user restarts normal computation using retry. *Except* is the pending exception term.

**break**(*PC*)

A break instruction is executed. *PC* is program counter. This port is used by the graphical debugger.

**cut\_call**(*PC*)

A cut is encountered at *PC*. This port is used by the graphical debugger to visualise the effect of the cut.

**cut\_exit**(*PC*)

A cut has been executed. See cut\_call(PC) for more information.

*Frame* is a reference to the current local stack frame, which can be examined using[prolog\_frame\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_frame_attribute/3). *Choice* is a reference to the last choice point and can be examined using [prolog\_choice\_attribute/3](http://www.swi-prolog.org/pldoc/man?predicate=prolog_choice_attribute/3). *Action* must be unified with a term that specifies how execution must continue. The following actions are defined:

**abort**

Abort execution. See [abort/0](http://www.swi-prolog.org/pldoc/man?predicate=abort/0).

**continue**

Continue (i.e., *creep* in the command line debugger).

**fail**

Make the current goal fail.

**ignore**

Step over the current goal without executing it.

**nodebug**

Continue execution in normal nodebugging mode. See [nodebug/0](http://www.swi-prolog.org/pldoc/man?predicate=nodebug/0).

**retry**

Retry the current frame.

**retry**(*Frame*)

Retry the given frame. This must be a parent of the current frame.

**skip**

Skip over the current goal (i.e., *skip* in the command line debugger).

**up**

Skip to the parent goal (i.e., *up* in the command line debugger).

Together with the predicates described in [section 4.37](http://www.swi-prolog.org/pldoc/man?section=debugger) and the other predicates of this chapter, this predicate enables the Prolog user to define a complete new debugger in Prolog. Besides this, it enables the Prolog programmer to monitor the execution of a program. The example below records all goals trapped by the tracer in the database.

prolog\_trace\_interception(Port, Frame, \_PC, continue) :-

prolog\_frame\_attribute(Frame, goal, Goal),

prolog\_frame\_attribute(Frame, level, Level),

recordz(trace, trace(Port, Level, Goal)).

To trace the execution of `go' this way the following query should be given:

?- trace, go, notrace.

|  |  |
| --- | --- |
| **project\_attributes/2** | Project constraints to query variables |

**prompt1**(*+Prompt*)

Sets the prompt for the next line to be read. Continuation lines will be read using the prompt defined by [prompt/2](http://www.swi-prolog.org/pldoc/man?predicate=prompt/2).

**prompt**(*-Old, +New*)

Set prompt associated with [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) and its derivatives. *Old* is first unified with the current prompt. On success the prompt will be set to *New* if this is an atom. Otherwise an error message is displayed. A prompt is printed if one of the read predicates is called and the cursor is at the left margin. It is also printed whenever a newline is given and the term has not been terminated. Prompts are only printed when the current input stream is *user*.

**protocol**(*+File*)

Start protocolling on file *File*. If there is already a protocol file open, then close it first. If *File*exists it is truncated.

**protocola**(*+File*)

Equivalent to [protocol/1](http://www.swi-prolog.org/pldoc/man?predicate=protocol/1), but does not truncate the *File* if it exists.

**protocolling**(*-File*)

True if a protocol was started with [protocol/1](http://www.swi-prolog.org/pldoc/man?predicate=protocol/1) or [protocola/1](http://www.swi-prolog.org/pldoc/man?predicate=protocola/1) and unifies *File* with the current protocol output file.

**public** *:PredicateIndicator, ...*

Instructs the cross-referencer that the predicate can be called. It has no semantics.69 The public declaration can be queried using [predicate\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=predicate_property/2). The [public/1](http://www.swi-prolog.org/pldoc/man?predicate=public/1) directive does *not*export the predicate (see [module/1](http://www.swi-prolog.org/pldoc/man?predicate=module/1) and [export/1](http://www.swi-prolog.org/pldoc/man?predicate=export/1)). The public directive is used for (1) direct calls into the module from, e.g., foreign code, (2) direct calls into the module from other modules, or (3) flag a predicate as being called if the call is generated by meta-calling constructs that are not analysed by the cross-referencer.

**put**(*+Char*)

Write *Char* to the current output stream. *Char* is either an integer expression evaluating to a character code or an atom of one character. Deprecated. New code should use [put\_char/1](http://www.swi-prolog.org/pldoc/man?predicate=put_char/1) or[put\_code/1](http://www.swi-prolog.org/pldoc/man?predicate=put_code/1).

**put**(*+Stream, +Char*)

Write *Char* to *Stream*. See [put/1](http://www.swi-prolog.org/pldoc/man?predicate=put/1) for details.

**put\_assoc**(*+Key, +Assoc, +Value, ?NewAssoc*)

*NewAssoc* is an association list identical to *Assoc* except that *Key* is associated with *Value*. This can be used to insert and change associations.

**put\_attr**(*+Var, +Module, +Value*)

If *Var* is a variable or attributed variable, set the value for the attribute named *Module* to *Value*. If an attribute with this name is already associated with *Var*, the old value is replaced. Backtracking will restore the old value (i.e., an attribute is a mutable term; see also [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3)). This predicate raises a representation error if *Var* is not a variable and a type error if *Module* is not an atom.

**put\_attrs**(*+Var, -Attributes*)

Set all attributes of *Var*. See [get\_attrs/2](http://www.swi-prolog.org/pldoc/man?predicate=get_attrs/2) for a description of *Attributes*.

**put\_byte**(*+Byte*)

Write a single byte to the output. *Byte* must be an integer between 0 and 255.

**put\_byte**(*+Stream, +Byte*)

Write a single byte to *Stream*. *Byte* must be an integer between 0 and 255.

**put\_char**(*+Char*)

Write a character to the current output, obeying the encoding defined for the current output stream. Note that this may raise an exception if the encoding of the output stream cannot represent *Char*.

**put\_char**(*+Stream, +Char*)

Write a character to *Stream*, obeying the encoding defined for *Stream*. Note that this may raise an exception if the encoding of *Stream* cannot represent *Char*.

**put\_code**(*+Code*)

Similar to [put\_char/1](http://www.swi-prolog.org/pldoc/man?predicate=put_char/1), but using a *character code*. *Code* is a non-negative integer. Note that this may raise an exception if the encoding of the output stream cannot represent *Code*.

**put\_code**(*+Stream, +Code*)

Same as [put\_code/1](http://www.swi-prolog.org/pldoc/man?predicate=put_code/1) but directing *Code* to *Stream*.

**put\_dict**(*+New, +DictIn, -DictOut*)

*DictOut* is a new dict created by replacing or adding key-value pairs from *New* to *Dict*. *New* is either a dict or a valid input for [dict\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=dict_create/3). This predicate is normally accessed using the functional notation. Below are some examples:

?- A = point{x:1, y:2}.put(\_{x:3}).

A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3]).

A = point{x:3, y:2}.

?- A = point{x:1, y:2}.put([x=3,z=0]).

A = point{x:3, y:2, z:0}.

**put\_dict**(*+Key, +DictIn, +Value, -DictOut*)

*DictOut* is a new dict created by replacing or adding *Key*-*Value* to *DictIn*. This predicate is normally accessed using the functional notation. Below is an example:

?- A = point{x:1, y:2}.put(x, 3).

A = point{x:3, y:2}.

**qcompile**(*:File*)

Takes a file specification as [consult/1](http://www.swi-prolog.org/pldoc/man?predicate=consult/1), etc., and, in addition to the normal compilation, creates a *Quick Load File* from *File*. The file extension of this file is .qlf. The basename of the Quick Load File is the same as the input file.

If the file contains `:- consult(*+File*)', `:- [*+File*]' or `:- load\_files(*+File*, [qcompile(part), ...])' statements, the referred files are compiled into the same .qlf file. Other directives will be stored in the .qlf file and executed in the same fashion as when loading the .pl file.

For [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2), the same rules as described in [section 2.10](http://www.swi-prolog.org/pldoc/man?section=compilation) apply.

Conditional execution or optimisation may test the predicate [compiling/0](http://www.swi-prolog.org/pldoc/man?predicate=compiling/0).

Source references ([source\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=source_file/2)) in the Quick Load File refer to the Prolog source file from which the compiled code originates.

**qcompile**(*:File, +Options*)

As [qcompile/1](http://www.swi-prolog.org/pldoc/man?predicate=qcompile/1), but processes additional options as defined by [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2).bug

**qsave\_program**(*+File*)

Equivalent to qsave\_program(File, []).

**qsave\_program**(*+File, +Options*)

Saves the current state of the program to the file *File*. The result is a resource archive containing a saved state that expresses all Prolog data from the running program and all user-defined resources. Depending on the stand\_alone option, the resource is headed by the emulator, a Unix shell script or nothing. *Options* is a list of additional options:

**local**(*+KBytes*)

Limit for the local stack. See [section 2.4.3](http://www.swi-prolog.org/pldoc/man?section=cmdline).

**global**(*+KBytes*)

Limit for the global stack. See [section 2.4.3](http://www.swi-prolog.org/pldoc/man?section=cmdline).

**trail**(*+KBytes*)

Limit for the trail stack. See [section 2.4.3](http://www.swi-prolog.org/pldoc/man?section=cmdline).

**goal**(*:Callable*)

Initialization goal for the new executable (see **-g**).

**toplevel**(*:Callable*)

Top-level goal for the new executable (see **-t**).

**init\_file**(*+Atom*)

Default initialization file for the new executable. See **-f**.

**class**(*+Class*)

If runtime, only read resources from the state (default). If kernel, lock all predicates as system predicates. If development, save the predicates in their current state and keep reading resources from their source (if present). See also [resource/3](http://www.swi-prolog.org/pldoc/man?predicate=resource/3).

**autoload**(*+Boolean*)

If true (default), run [autoload/0](http://www.swi-prolog.org/pldoc/man?predicate=autoload/0) first.

**map**(*+File*)

Dump a human-readable trace of what has been saved in *File*.

**op**(*+Action*)

One of save (default) to save the current operator table or standard to use the initial table of the emulator.

**stand\_alone**(*+Boolean*)

If true, the emulator is the first part of the state. If the emulator is started it will test whether a boot file (state) is attached to the emulator itself and load this state. Provided the application has all libraries loaded, the resulting executable is completely independent of the runtime environment or location where it was built. See also [section 2.10.2.4](http://www.swi-prolog.org/pldoc/man?section=compilation).

**emulator**(*+File*)

File to use for the emulator. Default is the running Prolog image.

**foreign**(*+Action*)

If save, include shared objects (DLLs) into the saved state. See[current\_foreign\_library/2](http://www.swi-prolog.org/pldoc/man?predicate=current_foreign_library/2). If the program **strip** is available, this is first used to reduce the size of the shared object. If a state is started, [use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1) first tries to locate the foreign resource in the executable. When found it copies the content of the resource to a temporary file and loads it. If possible (Unix), the temporary object is deleted immediately after opening.164

**quasi\_quotation\_syntax**(*:SyntaxName*)

Declare the predicate *SyntaxName*/4 to implement the the quasi quote syntax *SyntaxName*. Normally used as a directive.

**quasi\_quotation\_syntax\_error**(*+Error*)

Report syntax\_error(Error) using the current location in the quasi quoted input parser.

**throws**

error(syntax\_error(Error), Position)

**random\_property**(*?Option*)

True when *Option* is a current property of the random generator. Currently, this predicate provides access to the state. This predicate is not present on systems where the state is inaccessible.

**state**(*-State*)

Describes the current state of the random generator. State is a normal Prolog term that can be asserted or written to a file. Applications should make no other assumptions about its representation. The only meaningful operation is to use as argument to[set\_random/1](http://www.swi-prolog.org/pldoc/man?predicate=set_random/1) using the state(State) option.bug

**rational**(*@Term*)

True if *Term* is bound to a rational number. Rational numbers include integers.

**rational**(*@Term, -Numerator, -Denominator*)

True if *Term* is a rational number with given *Numerator* and *Denominator*. The *Numerator* and*Denominator* are in canonical form, which means *Denominator* is a positive integer and there are no common divisors between *Numerator* and *Denominator*.

**read**(*-Term*)

Read the next Prolog term from the current input stream and unify it with *Term*. On a syntax error [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) displays an error message, attempts to skip the erroneous term and fails. On reaching end-of-file *Term* is unified with the atom end\_of\_file.

**read**(*+Stream, -Term*)

Read *Term* from *Stream*.

**read\_clause**(*+Stream, -Term, +Options*)

Equivalent to [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3), but sets options according to the current compilation context and optionally processes comments. Defined options:

**syntax\_errors**(*+Atom*)

See [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3), but the default is dec10 (report and restart).

**term\_position**(*-TermPos*)

Same as for [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3).

**subterm\_positions**(*-TermPos*)

Same as for [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3).

**variable\_names**(*-Bindings*)

Same as for [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3).

**process\_comment**(*+Boolean*)

If true (default), call prolog:comment\_hook(Comments, TermPos, Term) if this multifile hook is defined (see prolog:comment\_hook/3). This is used to drive PlDoc.

**comments**(*-Comments*)

If provided, unify *Comments* with the comments encountered while reading *Term*. This option implies process\_comment(false).

The singletons option of [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3) is initialised from the active style-checking mode. Themodule option is initialised to the current compilation module (see [prolog\_load\_context/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_load_context/2)).

**read\_history**(*+Show, +Help, +Special, +Prompt, -Term, -Bindings*)

Similar to [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) using the option variable\_names, but allows for history substitutions.[read\_history/6](http://www.swi-prolog.org/pldoc/man?predicate=read_history/6) is used by the top level to read the user's actions. *Show* is the command the user should type to show the saved events. *Help* is the command to get an overview of the capabilities. *Special* is a list of commands that are not saved in the history. *Prompt* is the first prompt given. Continuation prompts for more lines are determined by [prompt/2](http://www.swi-prolog.org/pldoc/man?predicate=prompt/2). A %w in the prompt is substituted by the event number. See [section 2.7](http://www.swi-prolog.org/pldoc/man?section=history) for available substitutions.

SWI-Prolog calls [read\_history/6](http://www.swi-prolog.org/pldoc/man?predicate=read_history/6) as follows:

read\_history(h, '!h', [trace], '%w ?- ', Goal, Bindings)

**read\_link**(*+File, -Link, -Target*)

If *File* points to a symbolic link, unify *Link* with the value of the link and *Target* to the file the link is pointing to. *Target* points to a file, directory or non-existing entry in the file system, but never to a link. Fails if *File* is not a link. Fails always on systems that do not support symbolic links.

**read\_pending\_input**(*+StreamIn, -Codes, ?Tail*)

Read input pending in the input buffer of *StreamIn* and return it in the difference list *Codes*-*Tail*. That is, the available characters codes are used to create the list *Codes* ending in the tail*Tail*. This predicate is intended for efficient unbuffered copying and filtering of input coming from network connections or devices.

The following code fragment realises efficient non-blocking copying of data from an input to an output stream. The [at\_end\_of\_stream/1](http://www.swi-prolog.org/pldoc/man?predicate=at_end_of_stream/1) call checks for end-of-stream and fills the input buffer. Note that the use of a [get\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=get_code/2) and [put\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=put_code/2) based loop requires a[flush\_output/1](http://www.swi-prolog.org/pldoc/man?predicate=flush_output/1) call after *each* [put\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=put_code/2). The [copy\_stream\_data/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_stream_data/2) does not allow for inspection of the copied data and suffers from the same buffering issues.

copy(In, Out) :-

repeat,

( at\_end\_of\_stream(In)

-> !

; read\_pending\_input(In, Chars, []),

format(Out, '~s', [Chars]),

flush\_output(Out),

fail

).

**read\_string**(*+Stream, ?Length, -String*)

Read at most *Length* characters from *Stream* and return them in the string *String*. If *Length* is unbound, *Stream* is read to the end and *Length* is unified with the number of characters read.

**read\_string**(*+Stream, +SepChars, +PadChars, -Sep, -String*)

Read a string from *Stream*, providing functionality similar to [split\_string/4](http://www.swi-prolog.org/pldoc/man?predicate=split_string/4). The predicate performs the following steps:

1. Skip all characters that match *PadChars*
2. Read up to a character that matches *SepChars* or end of file
3. Discard trailing characters that match *PadChars* from the collected input
4. Unify *String* with a string created from the input and *Sep* with the separator character read. If input was terminated by the end of the input, *Sep* is unified with -1.

The predicate [read\_string/5](http://www.swi-prolog.org/pldoc/man?predicate=read_string/5) called repeatedly on an input until *Sep* is -1 (end of file) is equivalent to reading the entire file into a string and calling [split\_string/4](http://www.swi-prolog.org/pldoc/man?predicate=split_string/4), provided that*SepChars* and *PadChars* are not *partially overlapping*.131 Below are some examples:

% Read a line

read\_string(Input, "\n", "\r", End, String)

% Read a line, stripping leading and trailing white space

read\_string(Input, "\n", "\r\t ", End, String)

% Read upto , or ), unifying End with 0', or 0')

read\_string(Input, ",)", "\t ", End, String)

**read\_term**(*-Term, +Options*)

Read a term from the current input stream and unify the term with *Term*. The reading is controlled by options from the list of *Options*. If this list is empty, the behaviour is the same as for [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1). The options are upward compatible with Quintus Prolog. The argument order is according to the ISO standard. Syntax errors are always reported using exception-handling (see [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3)). Options:

**backquoted\_string**(*Bool*)

If true, read `...` to a string object (see [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)). The default depends on the Prolog flag [back\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:back_quotes).

**character\_escapes**(*Bool*)

Defines how to read \ escape sequences in quoted atoms. See the Prolog flag[character\_escapes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:character_escapes) in [current\_prolog\_flag/2](http://www.swi-prolog.org/pldoc/man?predicate=current_prolog_flag/2). (SWI-Prolog).

**comments**(*-Comments*)

Unify *Comments* with a list of *Position*-*Comment*, where *Position* is a stream position object (see [stream\_position\_data/3](http://www.swi-prolog.org/pldoc/man?predicate=stream_position_data/3)) indicating the start of a comment and *Comment* is a string object containing the text including delimiters of a comment. It returns all comments from where the [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) call started up to the end of the term read.

**cycles**(*Bool*)

If true (default false), re-instantiate templates as produced by the corresponding[write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) option. Note that the default is false to avoid misinterpretation of@(Template, Substutions), while the default of [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) is true because emitting cyclic terms without using the template construct produces an infinitely large term (read: it will generate an error after producing a huge amount of output).

**dotlists**(*Bool*)

If true (default false), read .(a,[]) as a list, even if lists are internally nor constructed using the dot as functor. This is primarily intended to read the output from[write\_canonical/1](http://www.swi-prolog.org/pldoc/man?predicate=write_canonical/1) from other Prolog systems. See [section 5.1](http://www.swi-prolog.org/pldoc/man?section=ext-lists).

**double\_quotes**(*Atom*)

Defines how to read " ... " strings. See the Prolog flag [double\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:double_quotes). (SWI-Prolog).

**module**(*Module*)

Specify *Module* for operators, [character\_escapes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:character_escapes) flag and [double\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:double_quotes) flag. The value of the latter two is overruled if the corresponding [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3) option is provided. If no module is specified, the current `source module' is used. (SWI-Prolog).

**quasi\_quotations**(*-List*)

If present, unify *List* with the quasi quotations (see [section A.26](http://www.swi-prolog.org/pldoc/man?section=quasiquotations)) instead of evaluating quasi quotations. Each quasi quotation is a term quasi\_quotation(+Syntax, +Quotation, +VarDict, -Result), where *Syntax* is the term in {|Syntax||..|},*Quotation* is a list of character codes that represent the quotation, *VarDict* is a list of*Name*=*Variable* and *Result* is a variable that shares with the place where the quotation must be inserted. This option is intended to support tools that manipulate Prolog source text.

**singletons**(*Vars*)

As variable\_names, but only reports the variables occurring only once in the *Term* read. Variables starting with an underscore (`\_') are not included in this list. (ISO). If *Vars* is the constant warning, singleton variables are reported using [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). The variables appear in the order they have been read.

**syntax\_errors**(*Atom*)

If error (default), throw an exception on a syntax error. Other values are fail, which causes a message to be printed using [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2), after which the predicate fails,quiet which causes the predicate to fail silently, and dec10 which causes syntax errors to be printed, after which read\_term/[2,3] continues reading the next term. Using dec10,read\_term/[2,3] never fails. (Quintus, SICStus).

**subterm\_positions**(*TermPos*)

Describes the detailed layout of the term. The formats for the various types of terms are given below. All positions are character positions. If the input is related to a normal stream, these positions are relative to the start of the input; when reading from the terminal, they are relative to the start of the term.

***From*-*To***

Used for primitive types (atoms, numbers, variables).

**string\_position**(*From, To*)

Used to indicate the position of a string enclosed in double quotes (").

**brace\_term\_position**(*From, To, Arg*)

Term of the form {...}, as used in DCG rules. *Arg* describes the argument.

**list\_position**(*From, To, Elms, Tail*)

A list. *Elms* describes the positions of the elements. If the list specifies the tail as |<TailTerm> , *Tail* is unified with the term position of the tail, otherwise with the atom none.

**term\_position**(*From, To, FFrom, FTo, SubPos*)

Used for a compound term not matching one of the above. *FFrom* and *FTo*describe the position of the functor. *SubPos* is a list, each element of which describes the term position of the corresponding subterm.

**map\_position**(*From, To, TypeFrom, TypeTo, KeyValuePosList*)

Used for a map (see [section 5.4](http://www.swi-prolog.org/pldoc/man?section=dicts)). The position of the key-value pairs is described by *KeyValuePosList*, which is a list of **key\_value\_position/7** terms. The**key\_value\_position/7** terms appear in the order of the input. Because maps to not preserve ordering, the key is provided in the position description.

**key\_value\_position**(*From, To, SepFrom, SepTo, Key, KeyPos, ValuePos*)

Used for key-value pairs in a map (see [section 5.4](http://www.swi-prolog.org/pldoc/man?section=dicts)). It is similar to the**term\_position/5** that would be created, except that the key and value positions do not need an intermediate list and the key is provided in *Key* to enable synchronisation of the file position data with the data structure.

**term\_position**(*Pos*)

Unifies *Pos* with the starting position of the term read. *Pos* is of the same format as used by [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2).

**variables**(*Vars*)

Unify *Vars* with a list of variables in the term. The variables appear in the order they have been read. See also [term\_variables/2](http://www.swi-prolog.org/pldoc/man?predicate=term_variables/2). (ISO).

**variable\_names**(*Vars*)

Unify *Vars* with a list of `*Name* = *Var*', where *Name* is an atom describing the variable name and *Var* is a variable that shares with the corresponding variable in *Term*. (ISO). The variables appear in the order they have been read.

**read\_term**(*+Stream, -Term, +Options*)

Read term with options from *Stream*. See [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2).

**read\_term\_from\_atom**(*+Atom, -Term, +Options*)

Use [read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3) to read the next term from *Atom*. *Atom* is either an atom or a string object (see [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings)). It is not required for *Atom* to end with a full-stop. This predicate supersedes[atom\_to\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=atom_to_term/3).

**recorda**(*+Key, +Term*)

Equivalent to recorda(*Key*, *Term*, \_).

**recorda**(*+Key, +Term, -Reference*)

Assert *Term* in the recorded database under key *Key*. *Key* is a small integer (range[min\_tagged\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:min_tagged_integer) ...[max\_tagged\_integer](http://www.swi-prolog.org/pldoc/man?section=flags#flag:max_tagged_integer), atom or compound term. If the key is a compound term, only the name and arity define the key. *Reference* is unified with an opaque handle to the record (see [erase/1](http://www.swi-prolog.org/pldoc/man?predicate=erase/1)).

**recorded**(*+Key, -Value*)

Equivalent to recorded(*Key*, *Value*, \_).

**recorded**(*?Key, ?Value, ?Reference*)

True if *Value* is recorded under *Key* and has the given database *Reference*. If *Reference* is given, this predicate is semi-deterministic. Otherwise, it must be considered non-deterministic. If neither *Reference* nor *Key* is given, the triples are generated as in the code snippet below.65 See also [current\_key/1](http://www.swi-prolog.org/pldoc/man?predicate=current_key/1).

current\_key(Key),

recorded(Key, Value, Reference)

**recordz**(*+Key, +Term*)

Equivalent to recordz(*Key*, *Term*, \_).

**recordz**(*+Key, +Term, -Reference*)

Equivalent to [recorda/3](http://www.swi-prolog.org/pldoc/man?predicate=recorda/3), but puts the *Term* at the tail of the terms recorded under *Key*.

**redefine\_system\_predicate**(*+Head*)

This directive may be used both in module user and in normal modules to redefine any system predicate. If the system definition is redefined in module user, the new definition is the default definition for all sub-modules. Otherwise the redefinition is local to the module. The system definition remains in the module system.

Redefining system predicate facilitates the definition of compatibility packages. Use in other contexts is discouraged.

**reexport**(*+Files*)

Load and import predicates as [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) and re-export all imported predicates. The reexport declarations must immediately follow the module declaration.

**reexport**(*+File, +Import*)

Import from *File* as [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2) and re-export the imported predicates. The reexport declarations must immediately follow the module declaration.

**reload\_foreign\_libraries**

Reload all foreign libraries loaded (after restore of a state created using [qsave\_program/2](http://www.swi-prolog.org/pldoc/man?predicate=qsave_program/2).

**reload\_library\_index**

Force reloading the index after modifying the set of library directories by changing the rules for [library\_directory/1](http://www.swi-prolog.org/pldoc/man?predicate=library_directory/1), [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2), adding or deleting INDEX.pl files. This predicate does *not* update the INDEX.pl files. Check make\_library\_index/[1,2] and [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0) for updating the index files.

Normally, the index is reloaded automatically if a predicate cannot be found in the index and the set of library directories has changed. Using [reload\_library\_index/0](http://www.swi-prolog.org/pldoc/man?predicate=reload_library_index/0) is necessary if directories are removed or the order of the library directories is changed.

**rename\_file**(*+File1, +File2*)

Rename *File1* as *File2*. The semantics is compatible to the POSIX semantics of the rename() system call as far as the operating system allows. Notably, if *File2* exists, the operation succeeds (except for possible permission errors) and is *atomic* (meaning there is no window where *File2* does not exist).

**repeat**

Always succeed, provide an infinite number of choice points.

**require**(*+ListOfNameAndArity*)

Declare that this file/module requires the specified predicates to be defined ``with their commonly accepted definition''. This predicate originates from the Prolog portability layer for XPCE. It is intended to provide a portable mechanism for specifying that this module requires the specified predicates.

The implementation normally first verifies whether the predicate is already defined. If not, it will search the libraries and load the required library.

SWI-Prolog, having autoloading, does **not** load the library. Instead it creates a procedure header for the predicate if it does not exist. This will flag the predicate as `undefined'. See also [check/0](http://www.swi-prolog.org/pldoc/man?predicate=check/0) and [autoload/0](http://www.swi-prolog.org/pldoc/man?predicate=autoload/0).

**reset\_gensym**(*+Base*)

Restart generation of identifiers from *Base* at <*Base*>1. Used to make sure a program produces the same results on subsequent runs. Use with care.

**reset\_gensym**

Reset gensym for all registered keys. This predicate is available for compatibility only. New code is strongly advised to avoid the use of reset\_gensym or at least to reset only the keys used by your program to avoid unexpected side effects on other components.

**reset\_profiler**

Switches the profiler to false and clears all collected statistics.

**resource**(*+Name, +Class, +FileSpec*)

This predicate is defined as a dynamic predicate in the module user. Clauses for it may be defined in any module, including the user module. *Name* is the name of the resource (an atom). A resource name may contain any character, except for $ and :, which are reserved for internal usage by the resource library. *Class* describes the kind of object stored in the resource. In the current implementation, it is just an atom. *FileSpec* is a file specification that may exploit [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2) (see [absolute\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/2)).

Normally, resources are defined as unit clauses (facts), but the definition of this predicate also allows for rules. For proper generation of the saved state, it must be possible to enumerate the available resources by calling this predicate with all its arguments unbound.

Dynamic rules are useful to turn all files in a certain directory into resources, without specifying a resource for each file. For example, assume the [file\_search\_path/2](http://www.swi-prolog.org/pldoc/man?predicate=file_search_path/2) icons refers to the resource directory containing icon files. The following definition makes all these images available as resources:

resource(Name, image, icons(XpmName)) :-

atom(Name), !,

file\_name\_extension(Name, xpm, XpmName).

resource(Name, image, XpmFile) :-

var(Name),

absolute\_file\_name(icons(.), [type(directory)], Dir)

concat(Dir, '/\*.xpm', Pattern),

expand\_file\_name(Pattern, XpmFiles),

member(XpmFile, XpmFiles).

**retract**(*+Term*)

When *Term* is an atom or a term it is unified with the first unifying fact or clause in the database. The fact or clause is removed from the database.

**retractall**(*+Head*)

All facts or clauses in the database for which the *head* unifies with *Head* are removed. If *Head*refers to a predicate that is not defined, it is implicitly created as a dynamic predicate. See also[dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1).64

**same\_file**(*+File1, +File2*)

True if both filenames refer to the same physical file. That is, if *File1* and *File2* are the same string or both names exist and point to the same file (due to hard or symbolic links and/or relative vs. absolute paths). On systems that provide stat() with meaningful values for st\_devand st\_inode, [same\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=same_file/2) is implemented by comparing the device and inode identifiers. On Windows, [same\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=same_file/2) compares the strings returned by the GetFullPathName() system call.

**same\_term**(*@T1, @T2*)

True if *T1* and *T2* are equivalent and will remain equivalent, even if [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) is used on either of them. This means *T1* and *T2* are the same variable, equivalent atomic data or a compound term allocated at the same address.

**see**(*+SrcDest*)

Open *SrcDest* for reading and make it the current input (see [set\_input/1](http://www.swi-prolog.org/pldoc/man?predicate=set_input/1)). If *SrcDest* is a stream handle, just make this stream the current input. See the introduction of [section 4.16.3](http://www.swi-prolog.org/pldoc/man?section=IO)for details.

**seeing**(*?SrcDest*)

Same as [current\_input/1](http://www.swi-prolog.org/pldoc/man?predicate=current_input/1), except that user is returned if the current input is the streamuser\_input to improve compatibility with traditional Edinburgh I/O. See the introduction of[section 4.16.3](http://www.swi-prolog.org/pldoc/man?section=IO) for details.

**seek**(*+Stream, +Offset, +Method, -NewLocation*)

Reposition the current point of the given *Stream*. *Method* is one of bof, current or eof, indicating positioning relative to the start, current point or end of the underlying object.*NewLocation* is unified with the new offset, relative to the start of the stream.

Positions are counted in `units'. A unit is 1 byte, except for text files using 2-byte Unicode encoding (2 bytes) or *wchar* encoding (sizeof(wchar\_t)). The latter guarantees comfortable interaction with wide-character text objects. Otherwise, the use of [seek/4](http://www.swi-prolog.org/pldoc/man?predicate=seek/4) on non-binary files (see [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4)) is of limited use, especially when using multi-byte text encodings (e.g. UTF-8) or multi-byte newline files (e.g. DOS/Windows). On text files, SWI-Prolog offers reliable backup to an old position using [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) and [set\_stream\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream_position/2). Skipping *N* character codes is achieved calling [get\_code/2](http://www.swi-prolog.org/pldoc/man?predicate=get_code/2) *N* times or using [copy\_stream\_data/3](http://www.swi-prolog.org/pldoc/man?predicate=copy_stream_data/3), directing the output to a null stream (see [open\_null\_stream/1](http://www.swi-prolog.org/pldoc/man?predicate=open_null_stream/1)). If the seek modifies the current location, the line number and character position in the line are set to 0.

If the stream cannot be repositioned, a permission\_error is raised. If applying the offset would result in a file position less than zero, a domain\_error is raised. Behaviour when seeking to positions beyond the size of the underlying object depend on the object and possibly the operating system. The predicate [seek/4](http://www.swi-prolog.org/pldoc/man?predicate=seek/4) is compatible with Quintus Prolog, though the error conditions and signalling is ISO compliant. See also [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) and[set\_stream\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream_position/2).

**seen**

Close the current input stream. The new input stream becomes user\_input.

|  |  |
| --- | --- |
| **select\_dict/2** | Select matching attributes from a dict |

**select\_dict**(*+Select, +From, -Rest*)

True when the tags of *Select* and *From* have been unified, all keys in *Select* appear in *From* and the corresponding values have been unified. The key-value pairs of *From* that do not appear in *Select* are used to form an anonymous dict, which us unified with *Rest*. For example:

?- select\_dict(P{x:0, y:Y}, point{x:0, y:1, z:2}, R).

P = point,

Y = 1,

R = \_G1705{z:2}.

See also select\_dict/2 to ignore *Rest* and [>:</2](http://www.swi-prolog.org/pldoc/man?predicate=%3E%3A%3C/2) for a symmetric partial unification of two dicts.

**set\_end\_of\_stream**(*+Stream*)

Set the size of the file opened as *Stream* to the current file position. This is typically used in combination with the open-mode update.

**set\_flag**(*+Key, Value*)

Set flag *Key* to *Value*. Value must be an atom, small (64-bit) integer or float.

**set\_input**(*+Stream*)

Set the current input stream to become *Stream*. Thus, open(file, read, Stream), set\_input(Stream) is equivalent to see(file).

**set\_locale**(*+Locale*)

Set the default locale for the current thread, as well as the locale for the standard streams (user\_input, user\_output, user\_error, current\_output and current\_input. This locale is used for new streams, unless overruled using the locale(Locale) option of [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) or[set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2).

**set\_module**(*:Property*)

Modify properties of the module. Currently, the following properties may be modified:

**base**(*+Base*)

Set the default import module of the current module to *Module*. Typically, *Module* is one of user or system. See [section 6.9](http://www.swi-prolog.org/pldoc/man?section=importmodule).

**class**(*+Class*)

Set the class of the module. See [module\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=module_property/2).

**program\_space**(*+Bytes*)

Maximum amount of memory used to store the predicates defined inside the module. Raises a permission error if the current usage is above the requested limit. Setting the limit to 0 (zero) removes the limit. An attempt to assert clauses that causes the limit to be exceeded causes a resource\_error(program\_space) exception. See [assertz/1](http://www.swi-prolog.org/pldoc/man?predicate=assertz/1) and[module\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=module_property/2).

**set\_output**(*+Stream*)

Set the current output stream to become *Stream*. See also [with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2).

**set\_prolog\_IO**(*+In, +Out, +Error*)

Prepare the given streams for interactive behaviour normally associated to the terminal. *In*becomes the user\_input and current\_input of the calling thread. *Out* becomesuser\_output and current\_output. If *Error* equals *Out* an unbuffered stream is associated to the same destination and linked to user\_error. Otherwise *Error* is used for user\_error. Output buffering for *Out* is set to line and buffering on *Error* is disabled. See also [prolog/0](http://www.swi-prolog.org/pldoc/man?predicate=prolog/0)and [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2). The *clib* package provides the library library(prolog\_server), creating a TCP/IP server for creating an interactive session to Prolog.

**set\_prolog\_flag**(*:Key, +Value*)

Define a new Prolog flag or change its value. *Key* is an atom. If the flag is a system-defined flag that is not marked *changeable* above, an attempt to modify the flag yields apermission\_error. If the provided *Value* does not match the type of the flag, a type\_error is raised.

Some flags (e.g., [unknown](http://www.swi-prolog.org/pldoc/man?section=flags#flag:unknown)) are maintained on a per-module basis. The addressed module is determined by the *Key* argument.

In addition to ISO, SWI-Prolog allows for user-defined Prolog flags. The type of the flag is determined from the initial value and cannot be changed afterwards. Defined types areboolean (if the initial value is one of false, true, on or off), atom if the initial value is any other atom, integer if the value is an integer that can be expressed as a 64-bit signed value. Any other initial value results in an untyped flag that can represent any valid Prolog term.

The behaviour when *Key* denotes a non-existent key depends on the Prolog flag [user\_flags](http://www.swi-prolog.org/pldoc/man?section=flags#flag:user_flags). The default is to define them silently. New code is encouraged to use [create\_prolog\_flag/3](http://www.swi-prolog.org/pldoc/man?predicate=create_prolog_flag/3) for portability.

**set\_prolog\_stack**(*+Stack, +KeyValue*)

Set a parameter for one of the Prolog runtime stacks. *Stack* is one of local, global, trail orargument. The table below describes the *Key*(*Value*) pairs. *Value* can be an arithmetic integer expression. For example, to specify a 2 GB limit for the global stack, one can use:

?- set\_prolog\_stack(global, limit(2\*10\*\*9)).

Current settings can be retrieved with [prolog\_stack\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_stack_property/2).

**limit**(*+Bytes*)

Set the limit to which the stack is allowed to grow. If the specified value is lower than the current usage a permission\_error is raised. If the limit is larger than supported, the system silently reduces the requested limit to the system limit.

**min\_free**(*+Cells*)

Minimum amount of free space after trimming or shifting the stack. Setting this value higher can reduce the number of garbage collections and stack-shifts at the cost of higher memory usage. The spare stack amount is reported and specified in `cells'. A cell is 4 bytes in the 32-bit version and 8 bytes on the 64-bit version. See [address\_bits](http://www.swi-prolog.org/pldoc/man?section=flags#flag:address_bits). See also [trim\_stacks/0](http://www.swi-prolog.org/pldoc/man?predicate=trim_stacks/0) and [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0).

**spare**(*+Cells*)

All stacks trigger overflow before actually reaching the limit, so the resulting error can be handled gracefully. The spare stack is used for [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) from the garbage collector and for handling exceptions. The default suffices, unless the user redefines related hooks. Do **not** specify large values for this because it reduces the amount of memory available for your real task.

Related hooks are [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3) (redefining GC messages),[prolog\_trace\_interception/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_trace_interception/4) and [prolog\_exception\_hook/4](http://www.swi-prolog.org/pldoc/man?predicate=prolog_exception_hook/4).

**set\_random**(*+Option*)

Controls the random number generator accessible through the *functions* [random/1](http://www.swi-prolog.org/pldoc/man?function=random/1) and[random\_float/0](http://www.swi-prolog.org/pldoc/man?function=random_float/0). Note that the library library(random) provides an alternative API to the same random primitives.

**seed**(*+Seed*)

Set the seed of the random generator for this thread. *Seed* is an integer or the atomrandom. If random, repeat the initialization procedure described with the function[random/1](http://www.swi-prolog.org/pldoc/man?function=random/1). Here is an example:

?- set\_random(seed(111)), A is random(6).

A = 5.

?- set\_random(seed(111)), A is random(6).

A = 5.

**state**(*+State*)

Set the generator to a state fetched using the state property of [random\_property/1](http://www.swi-prolog.org/pldoc/man?predicate=random_property/1). Using other values may lead to undefined behaviour.106

**set\_stream**(*+Stream, +Attribute*)

Modify an attribute of an existing stream. *Attribute* specifies the stream property to set. If stream is a *pair* (see [stream\_pair/3](http://www.swi-prolog.org/pldoc/man?predicate=stream_pair/3)) both streams are modified, unless the property is only meaningful on one of the streams or setting both is not meaningful. In particular, eof\_actiononly applies to the *read* stream, representation\_errors only applies to the *write* stream and trying to set alias or line\_position on a pair results in a permission\_error exception. See also [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) and [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4).

**alias**(*AliasName*)

Set the alias of an already created stream. If *AliasName* is the name of one of the standard streams, this stream is rebound. Thus, set\_stream(S, current\_input) is the same as [set\_input/1](http://www.swi-prolog.org/pldoc/man?predicate=set_input/1), and by setting the alias of a stream to user\_input, etc., all user terminal input is read from this stream. See also [interactor/0](http://www.swi-prolog.org/pldoc/man?predicate=interactor/0).

**buffer**(*Buffering*)

Set the buffering mode of an already created stream. Buffering is one of full, line orfalse.

**buffer\_size**(*+Size*)

Set the size of the I/O buffer of the underlying stream to *Size* bytes.

**close\_on\_abort**(*Bool*)

Determine whether or not the stream is closed by [abort/0](http://www.swi-prolog.org/pldoc/man?predicate=abort/0). By default, streams are closed.

**close\_on\_exec**(*Bool*)

Set the close\_on\_exec property. See [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2).

**encoding**(*Atom*)

Defines the mapping between bytes and character codes used for the stream. See[section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for supported encodings. The value bom causes the stream to check whether the current character is a Unicode BOM marker. If a BOM marker is found, the encoding is set accordingly and the call succeeds. Otherwise the call fails.

**eof\_action**(*Action*)

Set end-of-file handling to one of eof\_code, reset or error.

**file\_name**(*FileName*)

Set the filename associated to this stream. This call can be used to set the file for error locations if *Stream* corresponds to *FileName* and is not obtained by opening the file directly but, for example, through a network service.

**line\_position**(*LinePos*)

Set the line position attribute of the stream. This feature is intended to correct position management of the stream after sending a terminal escape sequence (e.g., setting ANSI character attributes). Setting this attribute raises a permission error if the stream does not record positions. See [line\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=line_position/2) and [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) (property position).

**locale**(*+Locale*)

Change the locale of the stream. See [section 4.22](http://www.swi-prolog.org/pldoc/man?section=locale).

**newline**(*NewlineMode*)

Set input or output translation for newlines. See corresponding [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) for details. In addition to the detected modes, an input stream can be set in mode detect. It will be set to dos if a \r character was removed.

**timeout**(*Seconds*)

This option can be used to make streams generate an exception if it takes longer than*Seconds* before any new data arrives at the stream. The value *infinite* (default) makes the stream block indefinitely. Like [wait\_for\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=wait_for_input/3), this call only applies to streams that support the select() system call. For further information about timeout handling, see [wait\_for\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=wait_for_input/3). The exception is of the form

error(timeout\_error(read, Stream), \_)

**type**(*Type*)

Set the type of the stream to one of text or binary. See also [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) and the encodingproperty of streams. Switching to binary sets the encoding to octet. Switching to textsets the encoding to the default text encoding.

**record\_position**(*Bool*)

Do/do not record the line count and line position (see [line\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=line_count/2) and[line\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=line_position/2)).

**representation\_errors**(*Mode*)

Change the behaviour when writing characters to the stream that cannot be represented by the encoding. See also [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2) and [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars).

**tty**(*Bool*)

Modify whether Prolog thinks there is a terminal (i.e. human interaction) connected to this stream. On Unix systems the initial value comes from isatty(). On Windows, the initial user streams are supposed to be associated to a terminal. See also[stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2).

**set\_stream\_position**(*+Stream, +Pos*)

Set the current position of *Stream* to *Pos*. *Pos* is a term as returned by [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2)using the position(Pos) property. See also [seek/4](http://www.swi-prolog.org/pldoc/man?predicate=seek/4).

**setup\_call\_cleanup**(*:Setup, :Goal, :Cleanup*)

Calls (once(Setup), Goal). If *Setup* succeeds, *Cleanup* will be called exactly once after *Goal*is finished: either on failure, deterministic success, commit, or an exception. The execution of*Setup* is protected from asynchronous interrupts like [call\_with\_time\_limit/2](http://www.swi-prolog.org/pldoc/man?predicate=call_with_time_limit/2) (package clib) or[thread\_signal/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_signal/2). In most uses, *Setup* will perform temporary side-effects required by *Goal*that are finally undone by *Cleanup*.

Success or failure of *Cleanup* is ignored, and choice points it created are destroyed (as[once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1)). If *Cleanup* throws an exception, this is executed as normal.bug

Typically, this predicate is used to cleanup permanent data storage required to execute *Goal*, close file descriptors, etc. The example below provides a non-deterministic search for a term in a file, closing the stream as needed.

term\_in\_file(Term, File) :-

setup\_call\_cleanup(open(File, read, In),

term\_in\_stream(Term, In),

close(In) ).

term\_in\_stream(Term, In) :-

repeat,

read(In, T),

( T == end\_of\_file

-> !, fail

; T = Term

).

Note that it is impossible to implement this predicate in Prolog. The closest approximation would be to read all terms into a list, close the file and call [member/2](http://www.swi-prolog.org/pldoc/man?predicate=member/2). Without[setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3) there is no way to gain control if the choice point left by [repeat/0](http://www.swi-prolog.org/pldoc/man?predicate=repeat/0) is removed by a cut or an exception.

[setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3) can also be used to test determinism of a goal, providing a portable alternative to [deterministic/1](http://www.swi-prolog.org/pldoc/man?predicate=deterministic/1):

?- setup\_call\_cleanup(true,(X=1;X=2), Det=yes).

X = 1 ;

X = 2,

Det = yes ;

This predicate is under consideration for inclusion into the ISO standard. For compatibility with other Prolog implementations see [call\_cleanup/2](http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2).

**setup\_call\_catcher\_cleanup**(*:Setup, :Goal, +Catcher, :Cleanup*)

Similar to setup\_call\_cleanup(Setup, Goal, Cleanup) with additional information on the reason for calling *Cleanup*. Prior to calling *Cleanup*, *Catcher* unifies with the termination code (see below). If this unification fails, *Cleanup* is *not* called.

**exit**

*Goal* succeeded without leaving any choice points.

**fail**

*Goal* failed.

**!**

*Goal* succeeded with choice points and these are now discarded by the execution of a cut (or other pruning of the search tree such as if-then-else).

**exception**(*Exception*)

*Goal* raised the given *Exception*.

**external\_exception**(*Exception*)

*Goal* succeeded with choice points and these are now discarded due to an exception. For example:

?- setup\_call\_catcher\_cleanup(true, (X=1;X=2),

Catcher, writeln(Catcher)),

throw(ball).

external\_exception(ball)

ERROR: Unhandled exception: Unknown message: ball

**setarg**(*+Arg, +Term, +Value*)

Extra-logical predicate. Assigns the *Arg*-th argument of the compound term *Term* with the given *Value*. The assignment is undone if backtracking brings the state back into a position before the [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) call. See also [nb\_setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=nb_setarg/3).

This predicate may be used for destructive assignment to terms, using them as an extra-logical storage bin. Always try hard to avoid the use of [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) as it is not supported by many Prolog systems and one has to be very careful about unexpected copying as well as unexpected noncopying of terms. A good practice to improve somewhat on this situation is to make sure that terms whose arguments are subject to [setarg/3](http://www.swi-prolog.org/pldoc/man?predicate=setarg/3) have one unused and unshared variable in addition to the used arguments. This variable avoids unwanted sharing in, e.g., [copy\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/2), and causes the term to be considered as non-ground.

**setenv**(*+Name, +Value*)

Set an environment variable. *Name* and *Value* must be instantiated to atoms or integers. The environment variable will be passed to shell/[0-2] and can be requested using [getenv/2](http://www.swi-prolog.org/pldoc/man?predicate=getenv/2). They also influence [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2). Environment variables are shared between threads. Depending on the underlying C library, [setenv/2](http://www.swi-prolog.org/pldoc/man?predicate=setenv/2) and [unsetenv/1](http://www.swi-prolog.org/pldoc/man?predicate=unsetenv/1) may not be thread-safe and may cause memory leaks. Only changing the environment once and before starting threads is safe in all versions of SWI-Prolog.

**setlocale**(*+Category, -Old, +New*)

Set/Query the *locale* setting which tells the C library how to interpret text files, write numbers, dates, etc. Category is one of all, collate, ctype, messages, monetary, numeric or time. For details, please consult the C library locale documentation. See also [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars). Please note that the locale is shared between all threads and thread-safe usage of [setlocale/3](http://www.swi-prolog.org/pldoc/man?predicate=setlocale/3) is in general not possible. Do locale operations before starting threads or thoroughly study threading aspects of locale support in your environment before using in multithreaded environments. Locale settings are used by [format\_time/3](http://www.swi-prolog.org/pldoc/man?predicate=format_time/3), [collation\_key/2](http://www.swi-prolog.org/pldoc/man?predicate=collation_key/2) and [locale\_sort/2](http://www.swi-prolog.org/pldoc/man?predicate=locale_sort/2).

**setof**(*+Template, +Goal, -Set*)

Equivalent to [bagof/3](http://www.swi-prolog.org/pldoc/man?predicate=bagof/3), but sorts the result using [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2) to get a sorted list of alternatives without duplicates.

**shell**

Start an interactive Unix shell. Default is /bin/sh; the environment variable SHELL overrides this default. Not available for Win32 platforms.

**shell**(*+Command*)

Equivalent to `shell(Command, 0)'.

**shell**(*+Command, -Status*)

Execute *Command* on the operating system. *Command* is given to the Bourne shell (/bin/sh).*Status* is unified with the exit status of the command.

On Windows, shell/[1,2] executes the command using the CreateProcess() API and waits for the command to terminate. If the command ends with a & sign, the command is handed to the WinExec() API, which does not wait for the new task to terminate. See also [win\_exec/2](http://www.swi-prolog.org/pldoc/man?predicate=win_exec/2) and[win\_shell/2](http://www.swi-prolog.org/pldoc/man?predicate=win_shell/2). Please note that the CreateProcess() API does **not** imply the Windows command interpreter (**cmd.exe** and therefore commands that are built in the command interpreter can only be activated using the command interpreter. For example, a file can be compied using the command below.

?- shell('cmd.exe /C copy file1.txt file2.txt').

Note that many of the operations that can be achieved using the shell built-in commands can easily be achieved using Prolog primitives. See [make\_directory/1](http://www.swi-prolog.org/pldoc/man?predicate=make_directory/1), [delete\_file/1](http://www.swi-prolog.org/pldoc/man?predicate=delete_file/1),[rename\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=rename_file/2), etc. The clib package provides library(filesex), implementing various high level file operations such as [copy\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=copy_file/2). Using Prolog primitives instead of shell commands improves the portability of your program.

The library library(process) provides [process\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=process_create/3) and several related primitives that support more fine-grained interaction with processes, including I/O redirection and management of asynchronous processes.

**show\_profile**(*+Options*)

This predicate first calls prolog:show\_profile\_hook/1. If XPCE is loaded, this hook is used to activate a GUI interface to visualise the profile results. If not, a report is printed to the terminal according to *Options*:

**top**(*+N*)

Show the only top *N* predicates. Default is 25.

**cumulative**(*+Bool*)

If true (default false), include the time spent in children in the time reported for a predicate.

**size\_file**(*+File, -Size*)

Unify *Size* with the size of *File* in bytes.

**size\_nb\_set**(*+Set, -Size*)

Unify *Size* with the number of elements in *Set*.

**skip**(*+Code*)

Read the input until *Code* or the end of the file is encountered. A subsequent call to[get\_code/1](http://www.swi-prolog.org/pldoc/man?predicate=get_code/1) will read the first character after *Code*.

**skip**(*+Stream, +Code*)

Skip input (as [skip/1](http://www.swi-prolog.org/pldoc/man?predicate=skip/1)) on *Stream*.

**rl\_add\_history**(*+Line*)

Add a line to the Control-P/Control-N history system of the readline library.

**rl\_read\_history**(*+FileName*)

Read history from *FileName*, appending to the current history.

**rl\_read\_init\_file**(*+File*)

Read a readline initialisation file. Readline by default reads /.inputrc. This predicate may be used to read alternative readline initialisation files.

**rl\_write\_history**(*+FileName*)

Write current history to *FileName*. Can be used from [at\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=at_halt/1) to save the history.

**sleep**(*+Time*)

Suspend execution *Time* seconds. *Time* is either a floating point number or an integer. Granularity is dependent on the system's timer granularity. A negative time causes the timer to return immediately. On most non-realtime operating systems we can only ensure execution is suspended for **at least** *Time* seconds.

On Unix systems the [sleep/1](http://www.swi-prolog.org/pldoc/man?predicate=sleep/1) predicate is realised ---in order of preference--- by nanosleep(), usleep(), select() if the time is below 1 minute, or sleep(). On Windows systems Sleep() is used.

**sort**(*+List, -Sorted*)

True if *Sorted* can be unified with a list holding the elements of *List*, sorted to the standard order of terms (see [section 4.7](http://www.swi-prolog.org/pldoc/man?section=compare)). Duplicates are removed. The implementation is in C, using*natural merge sort*.111 The [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2) predicate can sort a cyclic list, returning a non-cyclic version with the same elements.

**sort**(*+Key, +Order, +List, -Sorted*)

True when *Sorted* can be unified with a list holding the element of *List*. *Key* determines which part of each element in *List* is used for comparing two term and *Order* describes the relation between each set of consecutive elements in *Sorted*.112

If *Key* is the integer zero (0), the entire term is used to compare two elements. Using *Key*=0 can be used to sort arbitrary Prolog terms. Other values for *Key* can only be used with compound terms or dicts (see [section 5.4](http://www.swi-prolog.org/pldoc/man?section=dicts)). An integer key extracts the *Key*-th argument from a compound term. An integer or atom key extracts the value from a dict that is associated with the given key. A type\_error is raised if the list element is of the wrong type and an existence\_error is raised if the compound has not enough argument or the dict does not contain the requested key.

Deeper nested elements of structures can be selected by using a list of keys for the *Key*argument.

The *Order* argument is described in the table below113

|  |  |  |
| --- | --- | --- |
| Order | Ordering | Duplicate handling |
| @< | ascending | remove |
| @=< | ascending | keep |
| @> | descending | remove |
| @>= | descending | keep |

The sort is *stable*, which implies that, if duplicates are kept, the order of duplicates is not changed. If duplicates are removed, only the first element of a sequence of duplicates appears in *Sorted*.

This predicate supersedes most of the other sorting primitives, for example:

sort(List, Sorted) :- sort(0, @<, List, Sorted).

msort(List, Sorted) :- sort(0, @=<, List, Sorted).

keysort(Pairs, Sorted) :- sort(1, @=<, Pairs, Sorted).

The following example sorts a list of rows, for example resulting from [csv\_read\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=csv_read_file/2)) ascending on the 3th column and descending on the 4th column:

sort(4, @>=, Rows0, Rows1),

sort(3, @=<, Rows1, Sorted).

See also [sort/2](http://www.swi-prolog.org/pldoc/man?predicate=sort/2) (ISO), [msort/2](http://www.swi-prolog.org/pldoc/man?predicate=msort/2), [keysort/2](http://www.swi-prolog.org/pldoc/man?predicate=keysort/2), [predsort/3](http://www.swi-prolog.org/pldoc/man?predicate=predsort/3) and [order\_by/2](http://www.swi-prolog.org/pldoc/man?predicate=order_by/2).

**source\_exports**(*+Spec, +Export*)

Is true if source *Spec* exports *Export*, a predicate indicator. Fails without error otherwise.

**source\_file**(*?File*)

True if *File* is a loaded Prolog source file. *File* is the absolute and canonical path to the source file.

**source\_file**(*:Pred, ?File*)

True if the predicate specified by *Pred* is owned by file *File*, where *File* is an absolute path name (see [absolute\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/2)). Can be used with any instantiation pattern, but the database only maintains the source file for each predicate. If *Pred* is a *multifile* predicate this predicate succeeds for all files that contribute clauses to *Pred*.42 See also [clause\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=clause_property/2). Note that the relation between files and predicates is more complicated if [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) is used. The predicate describes the *owner* of the predicate. See [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) for details.

**source\_file\_property**(*?File, ?Property*)

True when *Property* is a property of the loaded file *File*. If *File* is non-var, it can be a file specification that is valid for [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). Defined properties are:

**derived\_from**(*Original, OriginalModified*)

*File* was generated from the file *Original*, which was last modified at time*OriginalModified* at the time it was loaded. This property is available if *File* was loaded using the derived\_from(Original) option to [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2).

**includes**(*IncludedFile, IncludedFileModified*)

*File* used [include/1](http://www.swi-prolog.org/pldoc/man?predicate=include/1) to include *IncludedFile*. The last modified time of *IncludedFile* was*IncludedFileModified* at the time it was included.

**included\_in**(*MasterFile, Line*)

*File* was included into *MasterFile* from line *Line*. This is the inverse of the includesproperty.

**load\_context**(*Module, Location, Options*)

*Module* is the module into which the file was loaded. If *File* is a module, this is the module into which the exports are imported. Otherwise it is the module into which the clauses of the non-module file are loaded. *Location* describes the file location from which the file was loaded. It is either a term <*file*>:<*line*> or the atom user if the file was loaded from the terminal or another unknown source. *Options* are the options passed to [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). Note that all predicates to load files are mapped to [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2), using the option argument to specify the exact behaviour.

**load\_count**(*-Count*)

*Count* is the number of times the file have been loaded, i.e., 1 (one) if the file has been loaded once.

**modified**(*Stamp*)

File modification time when *File* was loaded. This is used by [make/0](http://www.swi-prolog.org/pldoc/man?predicate=make/0) to find files whose modification time is different from when it was loaded.

**module**(*Module*)

*File* is a module file that declares the module *Module*.

**number\_of\_clauses**(*Count*)

*Count* is the number of clauses associated with *File*. Note that clauses loaded from included files are counted as part of the main file.

**reloading**

Present if the file is currently being **re**loaded.

**source\_location**(*-File, -Line*)

If the last term has been read from a physical file (i.e., not from the file user or a string), unify*File* with an absolute path to the file and *Line* with the line number in the file. New code should use [prolog\_load\_context/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_load_context/2).

**split\_string**(*+String, +SepChars, +PadChars, -SubStrings*)

Break *String* into *SubStrings*. The *SepChars* argument provides the characters that act as separators and thus the length of *SubStrings* is one more than the number of separators found if *SepChars* and *PadChars* do not have common characters. If *SepChars* and *PadChars*are equal, sequences of adjacent separators act as a single separator. Leading and trailing characters for each substring that appear in *PadChars* are removed from the substring. The input arguments can be either atoms, strings or char/code lists. Compatible with ECLiPSe. Below are some examples:

% a simple split

?- split\_string("a.b.c.d", ".", "", L).

L = ["a", "b", "c", "d"].

% Consider sequences of separators as a single one

?- split\_string("/home//jan///nice/path", "/", "/", L).

L = ["home", "jan", "nice", "path"].

% split and remove white space

?- split\_string("SWI-Prolog, 7.0", ",", " ", L).

L = ["SWI-Prolog", "7.0"].

% only remove leading and trailing white space

?- split\_string(" SWI-Prolog ", "", "\s\t\n", L).

L = ["SWI-Prolog"].

In the typical use cases, *SepChars* either does not overlap *PadChars* or is equivalent to handle multiple adjacent separators as a single (often white space). The behaviour with partially overlapping sets of padding and separators should be considered undefined. See also[read\_string/5](http://www.swi-prolog.org/pldoc/man?predicate=read_string/5).

**spy**(*+Pred*)

Put a spy point on all predicates meeting the predicate specification *Pred*. See [section 4.5](http://www.swi-prolog.org/pldoc/man?section=listing).

**stamp\_date\_time**(*+TimeStamp, -DateTime, +TimeZone*)

Convert a *TimeStamp* to a *DateTime* in the given timezone. See [section 4.33.2.1](http://www.swi-prolog.org/pldoc/man?section=system) for details on the data types. *TimeZone* describes the timezone for the conversion. It is one of local to extract the local time, 'UTC' to extract a UTC time or an integer describing the seconds west of Greenwich.

**statistics**

Display a table of system statistics on the stream user\_error.

**statistics**(*+Key, -Value*)

Unify system statistics determined by *Key* with *Value*. The possible keys are given in the [table 6](http://www.swi-prolog.org/pldoc/statistics.html#tab:statistics). This predicate supports additional keys for compatibility reasons. These keys are described in [table 7](http://www.swi-prolog.org/pldoc/statistics.html#tab:qpstatistics).

|  |  |
| --- | --- |
| Native keys (times as float in seconds) | |
| agc | Number of atom garbage collections performed |
| agc\_gained | Number of atoms removed |
| agc\_time | Time spent in atom garbage collections |
| atoms | Total number of defined atoms |
| c\_stack | System (C-) stack limit. 0 if not known. |
| cgc | Number of clause garbage collections performed |
| cgc\_gained | Number of clauses reclaimed |
| cgc\_time | Time spent in clause garbage collections |
| clauses | Total number of clauses in the program |
| codes | Total size of (virtual) executable code in words |
| cputime | (User) CPU time since thread was started in seconds |
| epoch | Time stamp when thread was started |
| functors | Total number of defined name/arity pairs |
| global | Allocated size of the global stack in bytes |
| globalused | Number of bytes in use on the global stack |
| globallimit | Size to which the global stack is allowed to grow |
| global\_shifts | Number of global stack expansions |
| heapused | Bytes of heap in use by Prolog (0 if not maintained) |
| heap\_gc | Number of heap garbage collections performed. Only provided if SWI-Prolog is configured with Boehm-GC. See alsogarbage\_collect\_heap/0. |
| inferences | Total number of passes via the call and redo ports since Prolog was started |
| modules | Total number of defined modules |
| local | Allocated size of the local stack in bytes |
| local\_shifts | Number of local stack expansions |
| locallimit | Size to which the local stack is allowed to grow |
| localused | Number of bytes in use on the local stack |
| trail | Allocated size of the trail stack in bytes |
| trail\_shifts | Number of trail stack expansions |
| traillimit | Size to which the trail stack is allowed to grow |
| trailused | Number of bytes in use on the trail stack |
| shift\_time | Time spent in stack-shifts |
| stack | Total memory in use for stacks in all threads |
| predicates | Total number of predicates. This includes predicates that are undefined or not yet resolved. |
| process\_epoch | Time stamp when Prolog was started |
| process\_cputime | (User) CPU time since Prolog was started in seconds |
| thread\_cputime | MT-version: Seconds CPU time used by finished threads. Basically non-portable. Works on Linux, MacOSX, Windows and probably some more. |
| threads | MT-version: number of active threads |
| threads\_created | MT-version: number of created threads |

**Table 6 :** Keys for [statistics/2](http://www.swi-prolog.org/pldoc/man?predicate=statistics/2). Space is expressed in bytes. Time is expressed in seconds, represented as a floating point number.

|  |  |
| --- | --- |
| Compatibility keys (times in milliseconds) | |
| runtime | [ CPU time, CPU time since last ] (milliseconds, excluding time spent in garbage collection) |
| system\_time | [ System CPU time, System CPU time since last ] (milliseconds) |
| real\_time | [ Wall time, Wall time since last ] (integer seconds. See[get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1)) |
| walltime | [ Wall time since start, Wall time since last] (milliseconds, SICStus compatibility) |
| memory | [ Total unshared data, free memory ] (Uses getrusage() if available, otherwise incomplete own statistics.) |
| stacks | [ global use, local use ] |
| program | [ heap, 0 ] |
| global\_stack | [ global use, global free ] |
| local\_stack | [ local use, local free ] |
| trail | [ trail use, trail free ] |
| garbage\_collection | [ number of GC, bytes gained, time spent, bytes left ] The last column is a SWI-Prolog extension. It contains the sum of the memory left after each collection, which can be divided by the count to find the average working set size after GC. Use [Count, Gained, Time|\_] for compatiblity. |
| stack\_shifts | [ global shifts, local shifts, time spent ] |
| atoms | [ number, memory use, 0 ] |
| atom\_garbage\_collection | [ number of AGC, bytes gained, time spent ] |
| clause\_garbage\_collection | [ number of CGC, clauses gained, time spent ] |
| core | Same as memory |

**Table 7 :** Compatibility keys for [statistics/2](http://www.swi-prolog.org/pldoc/man?predicate=statistics/2). Time is expressed in milliseconds.

**stream\_pair**(*?StreamPair, ?Read, ?Write*)

This predicate can be used in mode (-,+,+) to create a *stream-pair* from an input stream and an output stream. Mode (+,-,-) can be used to get access to the underlying streams. If a stream has already been closed, the corresponding argument is left unbound. If mode (+,-,-) is used on a single stream, either *Read* or *Write* is unified with the stream while the other argument is left unbound. This behaviour simplifies writing code that must operate both on streams and stream pairs.

Stream-pairs can be used by all I/O operations on streams, where the operation selects the appropriate member of the pair. The predicate [close/1](http://www.swi-prolog.org/pldoc/man?predicate=close/1) closes the still open streams of the pair.75 The output stream is closed before the input stream. If closing the output stream results in an error, the input stream is still closed. Success is only returned if both streams were closed successfully.

**stream\_position\_data**(*?Field, +Pos, -Data*)

Extracts information from the opaque stream position term as returned by [stream\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=stream_property/2)requesting the position(Pos) property. *Field* is one of line\_count, line\_position,char\_count or byte\_count. See also [line\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=line_count/2), [line\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=line_position/2), [character\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=character_count/2) and[byte\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=byte_count/2).76

**stream\_property**(*?Stream, ?StreamProperty*)

True when *StreamProperty* is a property of *Stream*. If enumeration of streams or properties is demanded because either *Stream* or *StreamProperty* are unbound, the implementation enumerates all candidate streams and properties while locking the stream database. Properties are fetched without locking the stream and may be outdated before this predicate returns due to asynchronous activity.

**alias**(*Atom*)

If *Atom* is bound, test if the stream has the specified alias. Otherwise unify *Atom* with the first alias of the stream.bug

**buffer**(*Buffering*)

SWI-Prolog extension to query the buffering mode of this stream. *Buffering* is one offull, line or false. See also [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4).

**buffer\_size**(*Integer*)

SWI-Prolog extension to query the size of the I/O buffer associated to a stream in bytes. Fails if the stream is not buffered.

**bom**(*Bool*)

If present and true, a BOM (*Byte Order Mark*) was detected while opening the file for reading, or a BOM was written while opening the stream. See [section 2.18.1.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for details.

**close\_on\_abort**(*Bool*)

Determine whether or not [abort/0](http://www.swi-prolog.org/pldoc/man?predicate=abort/0) closes the stream. By default streams are closed.

**close\_on\_exec**(*Bool*)

Determine whether or not the stream is closed when executing a new process (exec() in Unix, CreateProcess() in Windows). Default is to close streams. This maps to fcntl()F\_SETFD using the flag FD\_CLOEXEC on Unix and (negated) HANDLE\_FLAG\_INHERIT on Windows.

**encoding**(*Encoding*)

Query the encoding used for text. See [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) for an overview of wide character and encoding issues in SWI-Prolog.

**end\_of\_stream**(*E*)

If *Stream* is an input stream, unify *E* with one of the atoms not, at or past. See alsoat\_end\_of\_stream/[0,1].

**eof\_action**(*A*)

Unify *A* with one of eof\_code, reset or error. See [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) for details.

**file\_name**(*Atom*)

If *Stream* is associated to a file, unify *Atom* to the name of this file.

**file\_no**(*Integer*)

If the stream is associated with a POSIX file descriptor, unify *Integer* with the descriptor number. SWI-Prolog extension used primarily for integration with foreign code. See also Sfileno() from SWI-Stream.h.

**input**

True if *Stream* has mode read.

**locale**(*Locale*)

True when *Locale* is the current locale associated with the stream. See [section 4.22](http://www.swi-prolog.org/pldoc/man?section=locale).

**mode**(*IOMode*)

Unify *IOMode* to the mode given to [open/4](http://www.swi-prolog.org/pldoc/man?predicate=open/4) for opening the stream. Values are: read,write, append and the SWI-Prolog extension update.

**newline**(*NewlineMode*)

One of posix or dos. If dos, text streams will emit \r\n for \n and discard \r from input streams. Default depends on the operating system.

**nlink**(*-Count*)

Number of hard links to the file. This expresses the number of `names' the file has. Not supported on all operating systems and the value might be bogus. See the documentation of fstat() for your OS and the value st\_nlink.

**output**

True if *Stream* has mode write, append or update.

**position**(*Pos*)

Unify *Pos* with the current stream position. A stream position is an opaque term whose fields can be extracted using [stream\_position\_data/3](http://www.swi-prolog.org/pldoc/man?predicate=stream_position_data/3). See also [set\_stream\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream_position/2).

**reposition**(*Bool*)

Unify *Bool* with *true* if the position of the stream can be set (see [seek/4](http://www.swi-prolog.org/pldoc/man?predicate=seek/4)). It is assumed the position can be set if the stream has a *seek-function* and is not based on a POSIX file descriptor that is not associated to a regular file.

**representation\_errors**(*Mode*)

Determines behaviour of character output if the stream cannot represent a character. For example, an ISO Latin-1 stream cannot represent Cyrillic characters. The behaviour is one of error (throw an I/O error exception), prolog (write \...\ escape code) or xml(write &#...; XML character entity). The initial mode is prolog for the user streams anderror for all other streams. See also [section 2.18.1](http://www.swi-prolog.org/pldoc/man?section=widechars) and [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2).

**timeout**(*-Time*)

*Time* is the timeout currently associated with the stream. See [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2) with the same option. If no timeout is specified, *Time* is unified to the atom infinite.

**type**(*Type*)

Unify *Type* with text or binary.

**tty**(*Bool*)

This property is reported with *Bool* equal to true if the stream is associated with a terminal. See also [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2).

**string**(*@Term*)

True if *Term* is bound to a string. Note that string here refers to the built-in atomic type string as described in [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings). Starting with version 7, the syntax for a string object is text between double quotes, such as "hello".48 See also the Prolog flag [double\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:double_quotes).

**string\_concat**(*?String1, ?String2, ?String3*)

Similar to [atom\_concat/3](http://www.swi-prolog.org/pldoc/man?predicate=atom_concat/3), but the unbound argument will be unified with a string object rather than an atom. Also, if both *String1* and *String2* are unbound and *String3* is bound to text, it breaks *String3*, unifying the start with *String1* and the end with *String2* as append does with lists. Note that this is not particularly fast on long strings, as for each redo the system has to create two entirely new strings, while the list equivalent only creates a single new list-cell and moves some pointers around.

**string\_length**(*+String, -Length*)

Unify *Length* with the number of characters in *String*. This predicate is functionally equivalent to [atom\_length/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_length/2) and also accepts atoms, integers and floats as its first argument.

**string\_chars**(*?String, ?Chars*)

Bi-directional conversion between a string and a list of characters (one-character atoms). At least one of the two arguments must be instantiated.

**string\_codes**(*?String, ?Codes*)

Bi-directional conversion between a string and a list of character codes. At least one of the two arguments must be instantiated.

**string\_code**(*?Index, +String, ?Code*)

True when *Code* represents the character at the 1-based *Index* position in *String*. If *Index* is unbound the string is scanned from index 1. Raises a domain error if *Index* is negative. Fails silently if *Index* is zero or greater than the length of *String*. The mode string\_code(-,+,+) is deterministic if the searched-for *Code* appears only once in *String*. See also [sub\_string/5](http://www.swi-prolog.org/pldoc/man?predicate=sub_string/5).

**string\_lower**(*+String, LowerCase*)

Convert *String* to lower case and unify the result with *UpperCase*.

**string\_upper**(*+String, -UpperCase*)

Convert *String* to upper case and unify the result with *UpperCase*.

|  |  |
| --- | --- |
| [**string\_predicate/1**](http://www.swi-prolog.org/pldoc/check.html#string_predicate/1) | *(hook)* Predicate contains strings |
|  |  |

**strip\_module**(*+Term, -Module, -Plain*)

Used in module-transparent predicates or meta-predicates to extract the referenced module and plain term. If *Term* is a module-qualified term, i.e. of the format *Module*:*Plain*, *Module*and *Plain* are unified to these values. Otherwise, *Plain* is unified to *Term* and *Module* to the context module.

**style\_check**(*+Spec*)

Modify/query style checking options. *Spec* is one of the terms below or a list of these.

* +*Style* enables a style check
* -*Style* disables a style check
* ?(*Style*) queries a style check (note the brackets). If *Style* is unbound, all active style check options are returned on backtracking.

Loading a file using [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2) or one of its derived predicates reset the style checking options to their value before loading the file, scoping the option to the remainder of the file and all files loaded *after* changing the style checking.

**singleton**(*true*)

The predicate [read\_clause/3](http://www.swi-prolog.org/pldoc/man?predicate=read_clause/3) (used by the compiler to read source code) warns on variables appearing only once in a term (clause) which have a name not starting with an underscore. See [section 2.15.2.6](http://www.swi-prolog.org/pldoc/man?section=syntax) for details on variable handling and warnings.

**no\_effect**(*true*)

This warning is generated by the compiler for BIPs (built-in predicates) that are inlined by the compiler and for which the compiler can prove that they are meaningless. An example is using [==/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D%3D/2) against a not-yet-initialised variable as illustrated in the example below. This comparison is always false.

always\_false(X) :-

X == Y,

write(Y).

**var\_branches**(*false*)

Verifies that if a variable is introduced in a branch and used *after* the branch, it is introduced in all branches. This code aims at bugs following the skeleton below, wherep(Next) may be called with *Next* unbound.

p(Arg) :-

( Cond

-> Next = value1

; true

),

p(Next).

If a variable *V* is intended to be left unbound, one can use V=\_. This construct is removed by the compiler and thus has no implications for the performance of your program.

This check was suggested together with *semantic* singleton checking. The SWI-Prolog libraries contain about a hundred clauses that are triggered by this style check. Unlike semantic singleton analysis, only a tiny fraction of these clauses proofed faulty. In most cases, the branches failing to bind the variable fail or raise an exception or the caller handles the case where the variable is unbound. The status of this style check is unclear. It might be removed in the future or it might be enhanced with a deeper analysis to be more precise.

**atom**(*true*)

The predicate [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1) and derived predicates produce an error message on quoted atoms or strings with more than 6 *unescaped* newlines. Newlines may be escaped with \or \c. This flag also enables warnings on \<*newline*> followed by blank space in native mode. See [section 2.15.2.1](http://www.swi-prolog.org/pldoc/man?section=syntax). Note that the ISO standard does not allow for unescaped newlines in quoted atoms.

**discontiguous**(*true*)

Warn if the clauses for a predicate are not together in the same source file. It is advised to disable the warning for discontiguous predicates using the [discontiguous/1](http://www.swi-prolog.org/pldoc/man?predicate=discontiguous/1) directive.

**charset**(*false*)

Warn on atoms and variable names holding non-ASCII characters that are not quoted. See also [section 2.15.1.1](http://www.swi-prolog.org/pldoc/man?section=syntax).

**sub\_atom**(*+Atom, ?Before, ?Len, ?After, ?Sub*)

ISO predicate for breaking atoms. It maintains the following relation: *Sub* is a sub-atom of*Atom* that starts at *Before*, has *Len* characters, and *Atom* contains *After* characters after the match.

?- sub\_atom(abc, 1, 1, A, S).

A = 1, S = b

The implementation minimises non-determinism and creation of atoms. This is a flexible predicate that can do search, prefix- and suffix-matching, etc.

**sub\_atom\_icasechk**(*+Haystack, ?Start, +Needle*)

True when *Needle* is a sub atom of *Haystack* starting at *Start*. The match is `half case insensitive', i.e., uppercase letters in *Needle* only match themselves, while lowercase letters in*Needle* match case insensitively. *Start* is the first 0-based offset inside *Haystack* where *Needle*matches.94

**sub\_string**(*+String, ?Before, ?Length, ?After, ?SubString*)

*SubString* is a substring of *String*. There are *Before* characters in *String* before *SubString*,*SubString* contains *Length* character and is followed by *After* characters in *String*. If not enough information is provided to compute the start of the match, *String* is scanned left-to-right. This predicate is functionally equivalent to [sub\_atom/5](http://www.swi-prolog.org/pldoc/man?predicate=sub_atom/5), but operates on strings. The following example splits a string of the form <*name*>=<*value*> into the name part (an atom) and the value (a string).

name\_value(String, Name, Value) :-

sub\_string(String, Before, \_, After, "="), !,

sub\_string(String, 0, Before, \_, NameString),

atom\_string(Name, NameString),

sub\_string(String, \_, After, 0, Value).

**subsumes\_term**(*@Generic, @Specific*)

True if *Generic* can be made equivalent to *Specific* by only binding variables in *Generic*. The current implementation performs the unification and ensures that the variable set of *Specific*is not changed by the unification. On success, the bindings are undone.54 This predicate respects constraints.

**succ**(*?Int1, ?Int2*)

True if *Int2 = Int1 + 1* and *Int1 >= 0*. At least one of the arguments must be instantiated to a natural number. This predicate raises the domain error not\_less\_than\_zero if called with a negative integer. E.g. succ(X, 0) fails silently and succ(X, -1) raises a domain error.97

**swritef**(*-String, +Format*)

Equivalent to swritef(String, Format, []).

**swritef**(*-String, +Format, +Arguments*)

Equivalent to [writef/2](http://www.swi-prolog.org/pldoc/man?predicate=writef/2), but ``writes'' the result on *String* instead of the current output stream. Example:

?- swritef(S, '%15L%w', ['Hello', 'World']).

S = "Hello World"

**tab**(*+Amount*)

Write *Amount* spaces on the current output stream. *Amount* should be an expression that evaluates to a positive integer (see [section 4.26](http://www.swi-prolog.org/pldoc/man?section=arith)).

**tab**(*+Stream, +Amount*)

Write *Amount* spaces to *Stream*.

**tdebug**

Call [tdebug/1](http://www.swi-prolog.org/pldoc/man?predicate=tdebug/1) in all running threads.

**tdebug**(*+ThreadId*)

Prepare *ThreadId* for debugging using the graphical tracer. This implies installing the tracer hooks in the thread and switching the thread to debug mode using [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0). The call is injected into the thread using [thread\_signal/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_signal/2). We refer to the documentation of this predicate for asynchronous interaction with threads. New threads created inherit their debug mode from the thread that created them.

**tell**(*+SrcDest*)

Open *SrcDest* for writing and make it the current output (see [set\_output/1](http://www.swi-prolog.org/pldoc/man?predicate=set_output/1)). If *SrcDest* is a stream handle, just make this stream the current output. See the introduction of [section 4.16.3](http://www.swi-prolog.org/pldoc/man?section=IO) for details.

**telling**(*?SrcDest*)

Same as [current\_output/1](http://www.swi-prolog.org/pldoc/man?predicate=current_output/1), except that user is returned if the current output is the streamuser\_output to improve compatibility with traditional Edinburgh I/O. See the introduction of[section 4.16.3](http://www.swi-prolog.org/pldoc/man?section=IO) for details.

**term\_expansion**(*+Term1, -Term2*)

Dynamic and multifile predicate, normally not defined. When defined by the user all terms read during consulting are given to this predicate. If the predicate succeeds Prolog will assert*Term2* in the database rather than the read term (*Term1*). *Term2* may be a term of the form ?- Goal. or :- Goal. *Goal* is then treated as a directive. If *Term2* is a list, all terms of the list are stored in the database or called (for directives). If *Term2* is of the form below, the system will assert *Clause* and record the indicated source location with it:

'$source\_location'(<*File*>, <*Line*>):<*Clause*>

When compiling a module (see [chapter 6](http://www.swi-prolog.org/pldoc/man?section=modules) and the directive [module/2](http://www.swi-prolog.org/pldoc/man?predicate=module/2)), [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2) will first try [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2) in the module being compiled to allow for term expansion rules that are local to a module. If there is no local definition, or the local definition fails to translate the term, [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2) will try [term\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=term_expansion/2) in module user. For compatibility with SICStus and Quintus Prolog, this feature should not be used. See also [expand\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_term/2),[goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2) and [expand\_goal/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_goal/2).

|  |  |
| --- | --- |
| [**term\_expansion/4**](http://www.swi-prolog.org/pldoc/consulting.html#term_expansion/4) | *(hook)* Convert term before compilation |

**term\_string**(*?Term, ?String*)

Bi-directional conversion between a term and a string. If *String* is instantiated, it is parsed and the result is unified with *Term*. Otherwise *Term* is `written' using the option quoted(true)and the result is converted to *String*.

**term\_string**(*?Term, ?String, +Options*)

As [term\_string/2](http://www.swi-prolog.org/pldoc/man?predicate=term_string/2), passing *Options* to either [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2) or [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2). For example:

?- term\_string(Term, 'a(A)', [variable\_names(VNames)]).

Term = a(\_G1466),

VNames = ['A'=\_G1466].

**term\_subsumer**(*+Special1, +Special2, -General*)

*General* is the most specific term that is a generalisation of *Special1* and *Special2*. The implementation can handle cyclic terms.

**term\_to\_atom**(*?Term, ?Atom*)

True if *Atom* describes a term that unifies with *Term*. When *Atom* is instantiated, *Atom* is parsed and the result unified with *Term*. If *Atom* has no valid syntax, a syntax\_errorexception is raised. Otherwise *Term* is ``written'' on *Atom* using [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) with the optionquoted(true). See also [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3), [with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2) and [term\_string/2](http://www.swi-prolog.org/pldoc/man?predicate=term_string/2).

**thread\_at\_exit**(*:Goal*)

Run *Goal* just before releasing the thread resources. This is to be compared to [at\_halt/1](http://www.swi-prolog.org/pldoc/man?predicate=at_halt/1), but only for the current thread. These hooks are run regardless of why the execution of the thread has been completed. When these hooks are run, the return code is already available through[thread\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_property/2) using the result of [thread\_self/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_self/1) as thread identifier. Note that there are two scenarios for using exit hooks. Using [thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) is typically used if the thread creates a side-effect that must be reverted if the thread dies. Another scenario is where the creator of the thread wants to be informed when the thread ends. That cannot be guaranteed by means of [thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) because it is possible that the thread cannot be created or dies almost instantly due to a signal or resource error. The at\_exit(Goal) option of[thread\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_create/3) is designed to deal with this scenario.

**thread\_create**(*:Goal, -Id, +Options*)

Create a new Prolog thread (and underlying C thread) and start it by executing *Goal*. If the thread is created successfully, the thread identifier of the created thread is unified to *Id*.*Options* is a list of options. The currently defined options are below. Stack size options can also take the value inf or infinite, which is mapped to the maximum stack size supported by the platform.

**alias**(*AliasName*)

Associate an `alias name' with the thread. This name may be used to refer to the thread and remains valid until the thread is joined (see [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2)).

**at\_exit**(*:AtExit*)

Register *AtExit* as using [thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) before entering the thread goal. Unlike calling[thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) as part of the normal *Goal*, this *ensures* the *AtExit* is called. Using[thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1), the thread may be signalled or run out of resources before[thread\_at\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_at_exit/1) is reached.

**debug**(*+Bool*)

Enable/disable debugging the new thread. If false (default true), the new thread is created with the property debug(false) and debugging is disabled before the new thread is started. The thread debugging predicates such as [tspy/1](http://www.swi-prolog.org/pldoc/man?predicate=tspy/1) and [tdebug/0](http://www.swi-prolog.org/pldoc/man?predicate=tdebug/0) do not signal threads with the debug property set to false.144

**detached**(*Bool*)

If false (default), the thread can be waited for using [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2). [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2) must be called on this thread to reclaim all resources associated with the thread. If true, the system will reclaim all associated resources automatically after the thread finishes. Please note that thread identifiers are freed for reuse after a detached thread finishes or a normal thread has been joined. See also [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2) and [thread\_detach/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_detach/1).

If a detached thread dies due to failure or exception of the initial goal, the thread prints a message using [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2). If such termination is considered normal, the code must be wrapped using [ignore/1](http://www.swi-prolog.org/pldoc/man?predicate=ignore/1) and/or [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) to ensure successful completion.

**inherit\_from**(*+ThreadId*)

Inherit defaults from the given *ThreadId* instead of the calling thread. This option was added to ensure that the \_\_thread\_pool\_manager (see [thread\_create\_in\_pool/4](http://www.swi-prolog.org/pldoc/man?predicate=thread_create_in_pool/4)), which is created lazily, has a predictable state. The following properties are inherited:

* The prompt (see [prompt/2](http://www.swi-prolog.org/pldoc/man?predicate=prompt/2))
* The *typein* module (see [module/1](http://www.swi-prolog.org/pldoc/man?predicate=module/1))
* The standard streams (user\_input, etc.)
* The default encoding (see [encoding](http://www.swi-prolog.org/pldoc/man?section=flags#flag:encoding))
* The default locale (see setlocale/1)
* All prolog flags
* The limits of Prolog stacks (see [set\_prolog\_stack/2](http://www.swi-prolog.org/pldoc/man?predicate=set_prolog_stack/2))

**global**(*K-Bytes*)

Set the limit to which the global stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the **-G** command line option.

**local**(*K-Bytes*)

Set the limit to which the local stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the **-L** command line option.

**c\_stack**(*K-Bytes*)

Set the limit to which the system stack of this thread may grow. The default, minimum and maximum values are system-dependent.145

**trail**(*K-Bytes*)

Set the limit to which the trail stack of this thread may grow. If omitted, the limit of the calling thread is used. See also the **-T** command line option.

The *Goal* argument is *copied* to the new Prolog engine. This implies that further instantiation of this term in either thread does not have consequences for the other thread: Prolog threads do not share data from their stacks.

**thread\_detach**(*+Id*)

Switch thread into detached state (see detached(Bool) option at [thread\_create/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_create/3)) at runtime.*Id* is the identifier of the thread placed in detached state. This may be the result of[thread\_self/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_self/1).

One of the possible applications is to simplify debugging. Threads that are created as*detached* leave no traces if they crash. For non-detached threads the status can be inspected using [thread\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_property/2). Threads nobody is waiting for may be created normally and detach themselves just before completion. This way they leave no traces on normal completion and their reason for failure can be inspected.

**thread\_exit**(*+Term*)

Terminates the thread immediately, leaving exited(Term) as result state for [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2). If the thread has the attribute detached(true) it terminates, but its exit status cannot be retrieved using [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2), making the value of *Term* irrelevant. The Prolog stacks and C thread are reclaimed.

The current implementation does not guarantee proper releasing of all mutexes and proper cleanup in [setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3), etc. Please use the exception mechanism ([throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1)) to abort execution using non-standard control.

**thread\_get\_message**(*?Term*)

Examines the thread message queue and if necessary blocks execution until a term that unifies to *Term* arrives in the queue. After a term from the queue has been unified to *Term*, the term is deleted from the queue.

Please note that non-unifying messages remain in the queue. After the following has been executed, thread 1 has the term b(gnu) in its queue and continues execution using *A* = gnat.

<thread 1>

thread\_get\_message(a(A)),

<thread 2>

thread\_send\_message(Thread\_1, b(gnu)),

thread\_send\_message(Thread\_1, a(gnat)),

See also [thread\_peek\_message/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_peek_message/1).

**thread\_get\_message**(*+Queue, ?Term*)

As [thread\_get\_message/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/1), operating on a given queue. It is allowed (but not advised) to get messages from the queue of other threads. This predicate raises an existence error exception if *Queue* doesn't exist or is destroyed using [message\_queue\_destroy/1](http://www.swi-prolog.org/pldoc/man?predicate=message_queue_destroy/1) while this predicate is waiting.

**thread\_get\_message**(*+Queue, ?Term, +Options*)

As [thread\_get\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/2), but providing additional *Options*:

**deadline**(*+AbsTime*)

The call fails (silently) if no message has arrived before *AbsTime*. See [get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1) for the representation of absolute time. If *AbsTime* is earlier then the current time,[thread\_get\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/3) fails immediately. Both resolution and maximum wait time is platform-dependent.150

**timeout**(*+Time*)

*Time* is a float or integer and specifies the maximum time to wait in seconds. This is a relative-time version of the deadline option. If both options are provided, the earlier time is effective.

If *Time* is 0 or 0.0, [thread\_get\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/3) examines the queue but does not suspend if no matching term is available. Note that unlike [thread\_peek\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_peek_message/2), a matching term is removed from the queue.

If *Time* *< 0*, [thread\_get\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/3) fails immediately without removing any message from the queue.

**thread\_initialization**(*:Goal*)

Run *Goal* when thread is started. This predicate is similar to [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1), but is intended for initialization operations of the runtime stacks, such as setting global variables as described in[section 7.3](http://www.swi-prolog.org/pldoc/man?section=gvar). *Goal* is run on four occasions: at the call to this predicate, after loading a saved state, on starting a new thread and on creating a Prolog engine through the C interface. On loading a saved state, *Goal* is executed *after* running the [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1) hooks.

**thread\_join**(*+Id, -Status*)

Wait for the termination of the thread with the given *Id*. Then unify the result status of the thread with *Status*. After this call, *Id* becomes invalid and all resources associated with the thread are reclaimed. Note that threads with the attribute detached(true) cannot be joined. See also [thread\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_property/2).

A thread that has been completed without [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2) being called on it is partly reclaimed: the Prolog stacks are released and the C thread is destroyed. A small data structure representing the exit status of the thread is retained until [thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2) is called on the thread. Defined values for *Status* are:

**true**

The goal has been proven successfully.

**false**

The goal has failed.

**exception**(*Term*)

The thread is terminated on an exception. See [print\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=print_message/2) to turn system exceptions into readable messages.

**exited**(*Term*)

The thread is terminated on [thread\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_exit/1) using the argument *Term*.

**thread\_local** *+Functor/+Arity, ...*

This directive is related to the [dynamic/1](http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1) directive. It tells the system that the predicate may be modified using [assert/1](http://www.swi-prolog.org/pldoc/man?predicate=assert/1), [retract/1](http://www.swi-prolog.org/pldoc/man?predicate=retract/1), etc., during execution of the program. Unlike normal shared dynamic data, however, each thread has its own clause list for the predicate. As a thread starts, this clause list is empty. If there are still clauses when the thread terminates, these are automatically reclaimed by the system (see also [volatile/1](http://www.swi-prolog.org/pldoc/man?predicate=volatile/1)). The thread\_local property implies the properties *dynamic* and *volatile*.

Thread-local dynamic predicates are intended for maintaining thread-specific state or intermediate results of a computation.

It is not recommended to put clauses for a thread-local predicate into a file, as in the example below, because the clause is only visible from the thread that loaded the source file. All other threads start with an empty clause list.

:- thread\_local

foo/1.

foo(gnat).

**DISCLAIMER** Whether or not this declaration is appropriate in the sense of the proper mechanism to reach the goal is still debated. If you have strong feelings in favour or against, please share them in the SWI-Prolog mailing list.

**thread\_message\_hook**(*+Term, +Kind, +Lines*)

As [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3), but this predicate is local to the calling thread (see [thread\_local/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_local/1)). This hook is called *before* [message\_hook/3](http://www.swi-prolog.org/pldoc/man?predicate=message_hook/3). The `pre-hook' is indented to catch messages they may be produced by calling some goal without affecting other threads.

**thread\_peek\_message**(*?Term*)

Examines the thread message queue and compares the queued terms with *Term* until one unifies or the end of the queue has been reached. In the first case the call succeeds, possibly instantiating *Term*. If no term from the queue unifies, this call fails. I.e.,[thread\_peek\_message/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_peek_message/1) never waits and does not remove any term from the queue. See also[thread\_get\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/3).

**thread\_peek\_message**(*+Queue, ?Term*)

As [thread\_peek\_message/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_peek_message/1), operating on a given queue. It is allowed to peek into another thread's message queue, an operation that can be used to check whether a thread has swallowed a message sent to it.

**thread\_property**(*?Id, ?Property*)

True if thread *Id* has *Property*. Either or both arguments may be unbound, enumerating all relations on backtracking. Calling [thread\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_property/2) does not influence any thread. See also[thread\_join/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_join/2). For threads that have an alias name, this name is returned in *Id* instead of the opaque thread identifier. Defined properties are:

**alias**(*Alias*)

*Alias* is the alias name of thread *Id*.

**detached**(*Boolean*)

Current detached status of the thread.

**status**(*Status*)

Current status of the thread. *Status* is one of:

**running**

The thread is running. This is the initial status of a thread. Please note that threads waiting for something are considered running too.

**false**

The *Goal* of the thread has been completed and failed.

**true**

The *Goal* of the thread has been completed and succeeded.

**exited**(*Term*)

The *Goal* of the thread has been terminated using [thread\_exit/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_exit/1) with *Term* as argument. If the underlying native thread has exited (using pthread\_exit()) *Term* is unbound.

**exception**(*Term*)

The *Goal* of the thread has been terminated due to an uncaught exception (see[throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1) and [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3)).

**system\_thread\_id**(*Integer*)

Thread identifier used by the operating system for the calling thread. Not available on all OSes. This is the same as the Prolog flag [system\_thread\_id](http://www.swi-prolog.org/pldoc/man?section=flags#flag:system_thread_id) for the calling thread. Access to the system thread identifier can, on some systems, be used to gain additional control over or information about Prolog threads.

See also [thread\_statistics/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_statistics/3) to obtain resource usage information and[message\_queue\_property/2](http://www.swi-prolog.org/pldoc/man?predicate=message_queue_property/2) to get the number of queued messages for a thread.

**thread\_self**(*-Id*)

Get the Prolog thread identifier of the running thread. If the thread has an alias, the alias name is returned.

**thread\_send\_message**(*+QueueOrThreadId, +Term*)

Place *Term* in the given queue or default queue of the indicated thread (which can even be the message queue of itself, see [thread\_self/1](http://www.swi-prolog.org/pldoc/man?predicate=thread_self/1)). Any term can be placed in a message queue, but note that the term is copied to the receiving thread and variable bindings are thus lost. This call returns immediately.

If more than one thread is waiting for messages on the given queue and at least one of these is waiting with a partially instantiated *Term*, the waiting threads are *all* sent a wake-up signal, starting a rush for the available messages in the queue. This behaviour can seriously harm performance with many threads waiting on the same queue as all-but-the-winner perform a useless scan of the queue. If there is only one waiting thread or all waiting threads wait with an unbound variable, an arbitrary thread is restarted to scan the queue.147

**thread\_send\_message**(*+Queue, +Term, +Options*)

As [thread\_send\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/2), but providing additional *Options*. These are to deal with the case that the queue has a finite maximum size and is full: whereas [thread\_send\_message/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/2) will block until the queue has drained sufficiently to accept a new message,[thread\_send\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/3) can accept a time-out or deadline analogously to[thread\_get\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_get_message/3). The options are:

**deadline**(*+AbsTime*)

The call fails (silently) if no space has become available before *AbsTime*. See [get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1)for the representation of absolute time. If *AbsTime* is earlier then the current time,[thread\_send\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/3) fails immediately. Both resolution and maximum wait time is platform-dependent.148

**timeout**(*+Time*)

*Time* is a float or integer and specifies the maximum time to wait in seconds. This is a relative-time version of the deadline option. If both options are provided, the earlier time is effective.

If *Time* is 0 or 0.0, [thread\_send\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/3) examines the queue and sends the message if space is availabel, but does not suspend if no space is available, failing immediately instead.

If *Time* *< 0*, [thread\_send\_message/3](http://www.swi-prolog.org/pldoc/man?predicate=thread_send_message/3) fails immediately without sending the message.

**thread\_setconcurrency**(*-Old, +New*)

Determine the concurrency of the process, which is defined as the maximum number of concurrently active threads. `Active' here means they are using CPU time. This option is provided if the thread implementation provides pthread\_setconcurrency(). Solaris is a typical example of this family. On other systems this predicate unifies *Old* to 0 (zero) and succeeds silently.

**thread\_signal**(*+ThreadId, :Goal*)

Make thread *ThreadId* execute *Goal* at the first opportunity. In the current implementation, this implies at the first pass through the *Call port*. The predicate [thread\_signal/2](http://www.swi-prolog.org/pldoc/man?predicate=thread_signal/2) itself places*Goal* into the signalled thread's signal queue and returns immediately.

Signals (interrupts) do not cooperate well with the world of multithreading, mainly because the status of mutexes cannot be guaranteed easily. At the call port, the Prolog virtual machine holds no locks and therefore the asynchronous execution is safe.

*Goal* can be any valid Prolog goal, including [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1) to make the receiving thread generate an exception, and [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) to start tracing the receiving thread.

In the Windows version, the receiving thread immediately executes the signal if it reaches a Windows GetMessage() call, which generally happens if the thread is waiting for (user) input.

**thread\_statistics**(*+Id, +Key, -Value*)

Obtains statistical information on thread *Id* as [statistics/2](http://www.swi-prolog.org/pldoc/man?predicate=statistics/2) does in single-threaded applications. This call supports all keys of [statistics/2](http://www.swi-prolog.org/pldoc/man?predicate=statistics/2), although only stack sizes, cputime,inferences and epoch yield different values for each thread.146

**threads**

Lists all current threads and their status.

**throw**(*+Exception*)

Raise an exception. The system looks for the innermost [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) ancestor for which *Exception*unifies with the *Catcher* argument of the [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) call. See [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) for details.

ISO demands that [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1) make a copy of *Exception*, walk up the stack to a [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) call, backtrack and try to unify the copy of *Exception* with *Catcher*. SWI-Prolog delays backtracking until it actually finds a matching [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) goal. The advantage is that we can start the debugger at the first possible location while preserving the entire exception context if there is no matching [catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) goal. This approach can lead to different behaviour if *Goal* and *Catcher* of[catch/3](http://www.swi-prolog.org/pldoc/man?predicate=catch/3) call shared variables. We assume this to be highly unlikely and could not think of a scenario where this is useful.59

In addition to explicit calls to [throw/1](http://www.swi-prolog.org/pldoc/man?predicate=throw/1), many built-in predicates throw exceptions directly from C. If the *Exception* term cannot be copied due to lack of stack space, the following actions are tried in order:

1. If the exception is of the form error(Formal, ImplementationDefined), try to raise the exception without the *ImplementationDefined* part.
2. Try to raise error(resource\_error(stack), global).
3. Abort (see [abort/0](http://www.swi-prolog.org/pldoc/man?predicate=abort/0)).

If an exception is raised in a call-back from C (see [chapter 10](http://www.swi-prolog.org/pldoc/man?section=foreign)) and not caught in the same call-back, [PL\_next\_solution()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_next_solution) fails and the exception context can be retrieved using[PL\_exception()](http://www.swi-prolog.org/pldoc/man?CAPI=PL_exception).

**time**(*:Goal*)

Execute *Goal* just like [call/1](http://www.swi-prolog.org/pldoc/man?predicate=call/1) and print time used, number of logical inferences and the average number of *lips* (logical inferences per second). Note that SWI-Prolog counts the actual executed number of inferences rather than the number of passes through the call and redo ports of the theoretical 4-port model. If *Goal* is non-deterministic, print statistics for each solution, where the reported values are relative to the previous answer.

**time\_file**(*+File, -Time*)

Unify the last modification time of *File* with *Time*. *Time* is a floating point number expressing the seconds elapsed since Jan 1, 1970. See also convert\_time/[2,8] and [get\_time/1](http://www.swi-prolog.org/pldoc/man?predicate=get_time/1).

**tmp\_file**(*+Base, -TmpName*)

Create a name for a temporary file. *Base* is an identifier for the category of file. The *TmpName*is guaranteed to be unique. If the system halts, it will automatically remove all created temporary files. *Base* is used as part of the final filename. Portable applications should limit themselves to alphanumeric characters.

Because it is possible to guess the generated filename, attackers may create the filesystem entry as a link and possibly create a security issue. New code should use [tmp\_file\_stream/3](http://www.swi-prolog.org/pldoc/man?predicate=tmp_file_stream/3).

**tmp\_file\_stream**(*+Encoding, -FileName, -Stream*)

Create a temporary filename *FileName* and open it for writing in the given *Encoding*. *Encoding*is a text-encoding name or binary. *Stream* is the output stream. If the OS supports it, the created file is only accessible to the current user. If the OS supports it, the file is created using the open()-flag O\_EXCL, which guarantees that the file did not exist before this call. This predicate is a safe replacement of [tmp\_file/2](http://www.swi-prolog.org/pldoc/man?predicate=tmp_file/2). Note that in those cases where the temporary file is needed to store output from an external command, the file must be closed first. E.g., the following downloads a file from a URL to a temporary file and opens the file for reading (on Unix systems you can delete the file for cleanup after opening it for reading):

open\_url(URL, In) :-

tmp\_file\_stream(text, File, Stream),

close(Stream),

process\_create(curl, ['-o', File, URL], []),

open(File, read, In),

delete\_file(File). % Unix-only

Temporary files created using this call are removed if the Prolog process terminates*gracefully*. Calling [delete\_file/1](http://www.swi-prolog.org/pldoc/man?predicate=delete_file/1) using *FileName* removes the file and removes the entry from the administration of files-to-be-deleted.

**tnodebug**

Disable debugging in all threads.

**tnodebug**(*+ThreadId*)

Disable debugging thread *ThreadId*.

**told**

Close the current output stream. The new output stream becomes user\_output.

**tprofile**(*+ThreadId*)

Start collecting profile data in *ThreadId* and ask the user to hit <*return*> to stop the profiler. See [section 4.39](http://www.swi-prolog.org/pldoc/man?section=profile) for details on the execution profiler.

**trace**

Start the tracer. [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) itself cannot be seen in the tracer. Note that the Prolog top level treats[trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0) special; it means `trace the next goal'.

**trace**(*+Pred*)

Equivalent to trace(*Pred*, +all).

**trace**(*+Pred, +Ports*)

Put a trace point on all predicates satisfying the predicate specification *Pred*. *Ports* is a list of port names (call, redo, exit, fail). The atom all refers to all ports. If the port is preceded by a - sign, the trace point is cleared for the port. If it is preceded by a +, the trace point is set.

The predicate [trace/2](http://www.swi-prolog.org/pldoc/man?predicate=trace/2) activates debug mode (see [debug/0](http://www.swi-prolog.org/pldoc/man?predicate=debug/0)). Each time a port (of the 4-port model) is passed that has a trace point set, the goal is printed as with [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0). Unlike [trace/0](http://www.swi-prolog.org/pldoc/man?predicate=trace/0), however, the execution is continued without asking for further information. Examples:

|  |  |
| --- | --- |
| ?- trace(hello). | Trace all ports of hello with any arity in any module. |
| ?- trace(foo/2, +fail). | Trace failures of foo/2 in any module. |
| ?- trace(bar/1, -all). | Stop tracing bar/1. |

The predicate [debugging/0](http://www.swi-prolog.org/pldoc/man?predicate=debugging/0) shows all currently defined trace points.

**tracing**

True if the tracer is currently switched on. [tracing/0](http://www.swi-prolog.org/pldoc/man?predicate=tracing/0) itself cannot be seen in the tracer.

**trim\_stacks**

Release stack memory resources that are not in use at this moment, returning them to the operating system. It can be used to release memory resources in a backtracking loop, where the iterations require typically seconds of execution time and very different, potentially large, amounts of stack space. Such a loop can be written as follows:

loop :-

generator,

trim\_stacks,

potentially\_expensive\_operation,

stop\_condition, !.

The Prolog top-level loop is written this way, reclaiming memory resources after every user query.

**true**

Always succeed. The predicate [true/0](http://www.swi-prolog.org/pldoc/man?predicate=true/0) is translated into a single virtual machine instruction.

**tspy**(*:Spec*)

Set a spy point as [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1) and enable debugging in all threads using [tdebug/0](http://www.swi-prolog.org/pldoc/man?predicate=tdebug/0). Note that removing spy points can be done using [nospy/1](http://www.swi-prolog.org/pldoc/man?predicate=nospy/1). Disabling spy points in a specific thread is achieved by [tnodebug/1](http://www.swi-prolog.org/pldoc/man?predicate=tnodebug/1).

**tspy**(*:Spec, +ThreadId*)

Set a spy point as [spy/1](http://www.swi-prolog.org/pldoc/man?predicate=spy/1) and enable the thread for debugging using [tdebug/1](http://www.swi-prolog.org/pldoc/man?predicate=tdebug/1). Note that a spy point is a global flag on a predicate that is visible from all threads. Spy points are honoured in all threads that are in debug mode and ignored in threads that are in nodebug mode.

**tty\_get\_capability**(*+Name, +Type, -Result*)

Get the capability named *Name* from the termcap library. See termcap(5) for the capability names. *Type* specifies the type of the expected result, and is one of string, number or bool. String results are returned as an atom, number results as an integer, and bool results as the atom on or off. If an option cannot be found, this predicate fails silently. The results are only computed once. Successive queries on the same capability are fast.

**tty\_goto**(*+X, +Y*)

Goto position (*X*, *Y*) on the screen. Note that the predicates [line\_count/2](http://www.swi-prolog.org/pldoc/man?predicate=line_count/2) and [line\_position/2](http://www.swi-prolog.org/pldoc/man?predicate=line_position/2)will not have a well-defined behaviour while using this predicate.

**tty\_put**(*+Atom, +Lines*)

Put an atom via the termcap library function tputs(). This function decodes padding information in the strings returned by [tty\_get\_capability/3](http://www.swi-prolog.org/pldoc/man?predicate=tty_get_capability/3) and should be used to output these strings. *Lines* is the number of lines affected by the operation, or 1 if not applicable (as in almost all cases).

**tty\_size**(*-Rows, -Columns*)

Determine the size of the terminal. Platforms:

**Unix**

If the system provides *ioctl* calls for this, these are used and [tty\_size/2](http://www.swi-prolog.org/pldoc/man?predicate=tty_size/2) properly reflects the actual size after a user resize of the window. As a fallback, the system uses[tty\_get\_capability/3](http://www.swi-prolog.org/pldoc/man?predicate=tty_get_capability/3) using li and co capabilities. In this case the reported size reflects the size at the first call and is not updated after a user-initiated resize of the terminal.

**Windows**

Getting the size of the terminal is provided for **swipl-win.exe**. The requested value reflects the current size. For the multithreaded version the console that is associated with the user\_input stream is used.

**ttyflush**

Flush pending output on stream user. See also flush\_output/[0,1].

**unify\_with\_occurs\_check**(*+Term1, +Term2*)

As [=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D/2), but using *sound unification*. That is, a variable only unifies to a term if this term does not contain the variable itself. To illustrate this, consider the two queries below.

1 ?- A = f(A).

A = f(A).

2 ?- unify\_with\_occurs\_check(A, f(A)).

false.

The first statement creates a *cyclic term*, also called a *rational tree*. The second executes logically sound unification and thus fails. Note that the behaviour of unification through [=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D/2)as well as implicit unification in the head can be changed using the Prolog flag [occurs\_check](http://www.swi-prolog.org/pldoc/man?section=flags#flag:occurs_check).

The SWI-Prolog implementation of [unify\_with\_occurs\_check/2](http://www.swi-prolog.org/pldoc/man?predicate=unify_with_occurs_check/2) is cycle-safe and only guards against *creating* cycles, not against cycles that may already be present in one of the arguments. This is illustrated in the following two queries:

?- X = f(X), Y = X, unify\_with\_occurs\_check(X, Y).

X = Y, Y = f(Y).

?- X = f(X), Y = f(Y), unify\_with\_occurs\_check(X, Y).

X = Y, Y = f(Y).

Some other Prolog systems interpret [unify\_with\_occurs\_check/2](http://www.swi-prolog.org/pldoc/man?predicate=unify_with_occurs_check/2) as if defined by the clause below, causing failure on the above two queries. Direct use of [acyclic\_term/1](http://www.swi-prolog.org/pldoc/man?predicate=acyclic_term/1) is portable and more appropriate for such applications.

unify\_with\_occurs\_check(X,X) :- acyclic\_term(X).

**unifiable**(*@X, @Y, -Unifier*)

If *X* and *Y* can unify, unify *Unifier* with a list of *Var* = *Value*, representing the bindings required to make *X* and *Y* equivalent.55 This predicate can handle cyclic terms. Attributed variables are handled as normal variables. Associated hooks are *not* executed.

**unix**(*+Command*)

This predicate comes from the Quintus compatibility library and provides a partial implementation thereof. It provides access to some operating system features and unlike the name suggests, is not operating system specific. Defined *Command*'s are below.

**system**(*+Command*)

Equivalent to calling [shell/1](http://www.swi-prolog.org/pldoc/man?predicate=shell/1). Use for compatibility only.

**shell**(*+Command*)

Equivalent to calling [shell/1](http://www.swi-prolog.org/pldoc/man?predicate=shell/1). Use for compatibility only.

**shell**

Equivalent to calling [shell/0](http://www.swi-prolog.org/pldoc/man?predicate=shell/0). Use for compatibility only.

**cd**

Equivalent to calling [working\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=working_directory/2) to the expansion (see [expand\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=expand_file_name/2)) of~. For compatibility only.

**cd**(*+Directory*)

Equivalent to calling [working\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=working_directory/2). Use for compatibility only.

**argv**(*-Argv*)

Unify *Argv* with the list of command line arguments provided to this Prolog run. Please note that Prolog system arguments and application arguments are separated by --. Integer arguments are passed as Prolog integers, float arguments and Prolog floating point numbers and all other arguments as Prolog atoms. New applications should use the Prolog flag [argv](http://www.swi-prolog.org/pldoc/man?section=flags#flag:argv). See also the Prolog flag [argv](http://www.swi-prolog.org/pldoc/man?section=flags#flag:argv).

A stand-alone program could use the following skeleton to handle command line arguments. See also [section 2.10.2.4](http://www.swi-prolog.org/pldoc/man?section=compilation).

main :-

current\_prolog\_flag(argv, Argv),

append(\_PrologArgs, [--|AppArgs], Argv), !,

main(AppArgs).

**unknown**(*-Old, +New*)

Edinburgh-Prolog compatibility predicate, interfacing to the ISO Prolog flag [unknown](http://www.swi-prolog.org/pldoc/man?section=flags#flag:unknown). Values are trace (meaning error) and fail. If the [unknown](http://www.swi-prolog.org/pldoc/man?section=flags#flag:unknown) flag is set to warning, [unknown/2](http://www.swi-prolog.org/pldoc/man?predicate=unknown/2)reports the value as trace.

**unload\_file**(*+File*)

Remove all clauses loaded from *File*. If *File* loaded a module, clear the module's export list and disassociate it from the file. *File* is a canonical filename or a file indicator that is valid for[load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2).

This predicate should be used with care. The multithreaded nature of SWI-Prolog makes removing static code unsafe. Attempts to do this should be reserved for development or situations where the application can guarantee that none of the clauses associated to *File* are active.

**unload\_foreign\_library**(*+FileSpec*)

*[det]***unload\_foreign\_library**(*+FileSpec, +Exit:atom*)

Unload a *shared object* or *DLL*. After calling the *Exit* function, the shared object is removed from the process. The default exit function is composed from =uninstall\_=, followed by the file base-name.

**unsetenv**(*+Name*)

Remove an environment variable from the environment. Some systems lack the underlying unsetenv() library function. On these systems [unsetenv/1](http://www.swi-prolog.org/pldoc/man?predicate=unsetenv/1) sets the variable to the empty string.

**upcase\_atom**(*+AnyCase, -UpperCase*)

Converts, similar to [downcase\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=downcase_atom/2), an atom to uppercase.

**use\_foreign\_library**(*+FileSpec*)

*[det]***use\_foreign\_library**(*+FileSpec, +Entry:atom*)

Load and install a foreign library as [load\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=load_foreign_library/1),2 and register the installation using [initialization/2](http://www.swi-prolog.org/pldoc/man?predicate=initialization/2) with the option now. This is similar to using:

:- initialization(load\_foreign\_library(foreign(mylib))).

but using the [initialization/1](http://www.swi-prolog.org/pldoc/man?predicate=initialization/1) wrapper causes the library to be loaded *after* loading of the file in which it appears is completed, while [use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1) loads the library *immediately*. I.e. the difference is only relevant if the remainder of the file uses functionality of the C-library.

**use\_module**(*+Files*)

Load the file(s) specified with *Files* just like [ensure\_loaded/1](http://www.swi-prolog.org/pldoc/man?predicate=ensure_loaded/1). The files must all be module files. All exported predicates from the loaded files are imported into the module from which this predicate is called. This predicate is equivalent to [ensure\_loaded/1](http://www.swi-prolog.org/pldoc/man?predicate=ensure_loaded/1), except that it raises an error if *Files* are not module files.

The imported predicates act as *weak symbols* in the module into which they are imported. This implies that a local definition of a predicate overrides (clobbers) the imported definition. If the flag [warn\_override\_implicit\_import](http://www.swi-prolog.org/pldoc/man?section=flags#flag:warn_override_implicit_import) is true (default), a warning is printed. Below is an example of a module that uses library(lists), but redefines [flatten/2](http://www.swi-prolog.org/pldoc/man?predicate=flatten/2), giving it a totally different meaning:

:- module(shapes, []).

:- use\_module(library(lists)).

flatten(cube, square).

flatten(ball, circle).

Loading the above file prints the following message:

Warning: /home/janw/Bugs/Import/t.pl:5:

Local definition of shapes:flatten/2

overrides weak import from lists

This warning can be avoided by (1) using [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2) to only import the predicates from the lists library that are actually used in the `shapes' module, (2) using theexcept([flatten/2]) option of [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2), (3) use :- abolish([flatten/2](http://www.swi-prolog.org/pldoc/man?predicate=flatten/2)). before the local definition or (4) setting [warn\_override\_implicit\_import](http://www.swi-prolog.org/pldoc/man?section=flags#flag:warn_override_implicit_import) to false. Globally disabling this warning is only recommended if overriding imported predicates is common as a result of design choices or the program is ported from a system that silently overrides imported predicates.

Note that it is always an error to import two modules with [use\_module/1](http://www.swi-prolog.org/pldoc/man?predicate=use_module/1) that export the same predicate. Such conflicts must be resolved with [use\_module/2](http://www.swi-prolog.org/pldoc/man?predicate=use_module/2) as described above.

**use\_module**(*+File, +ImportList*)

Load *File*, which must be a module file, and import the predicates as specified by *ImportList*.*ImportList* is a list of predicate indicators specifying the predicates that will be imported from the loaded module. *ImportList* also allows for renaming or import-everything-except. See also the import option of [load\_files/2](http://www.swi-prolog.org/pldoc/man?predicate=load_files/2). The first example below loads [member/2](http://www.swi-prolog.org/pldoc/man?predicate=member/2) from the listslibrary and [append/2](http://www.swi-prolog.org/pldoc/man?predicate=append/2) under the name list\_concat, which is how this predicate is named in YAP. The second example loads all exports from library option except for [meta\_options/3](http://www.swi-prolog.org/pldoc/man?predicate=meta_options/3). These renaming facilities are generally used to deal with portability issues with as few changes as possible to the actual code. See also [section C](http://www.swi-prolog.org/pldoc/man?section=dialect) and [section 6.7](http://www.swi-prolog.org/pldoc/man?section=reexport).

:- use\_module(library(lists), [ member/2,

append/2 as list\_concat

]).

:- use\_module(library(option), except([meta\_options/3])).

|  |  |
| --- | --- |
| [**valid\_string\_goal/1**](http://www.swi-prolog.org/pldoc/check.html#valid_string_goal/1) | *(hook)* Goal handles strings |
|  |  |

**var**(*@Term*)

True if *Term* currently is a free variable.

**var\_number**(*@Term, -VarNumber*)

True if *Term* is numbered by [numbervars/3](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/3) and *VarNumber* is the number given to this variable. This predicate avoids the need for unification with '$VAR'(X) and opens the path for replacing this valid Prolog term by an internal representation that has no textual equivalent.

**var\_property**(*+Var, ?Property*)

True when *Property* is a property of *Var*. These properties are available during goal- and term-expansion. Defined properties are below. Future versions are likely to provide more properties, such as whether the variable is a singleton or whether the variable is referenced in the remainder of the term. See also [goal\_expansion/2](http://www.swi-prolog.org/pldoc/man?predicate=goal_expansion/2).

**fresh**(*Bool*)

Bool has the value *true* if the variable is guaranteed to be unbound at entry of the goal, otherwise its value is *false*. This implies that the variable first appears in this goal or a previous appearance was in a negation ([\+/1](http://www.swi-prolog.org/pldoc/man?predicate=%5C%2B/1)) or a different branch of a disjunction.

**name**(*Name*)

True when variable appears with the given name in the source.

**variant\_sha1**(*+Term, -SHA1*)

Compute a SHA1-hash from *Term*. The hash is represented as a 40-byte hexadecimal atom. Unlike [term\_hash/2](http://www.swi-prolog.org/pldoc/man?predicate=term_hash/2) and friends, this predicate produces a hash key for non-ground terms. The hash is invariant over variable-renaming (see [=@=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D@%3D/2)) and constants over different invocations of Prolog.bug

This predicate raises an exception when trying to compute the hash on a cyclic term or attributed term. Attributed terms are not handled because [subsumes\_chk/2](http://www.swi-prolog.org/pldoc/man?predicate=subsumes_chk/2) is not considered well defined for attributed terms. Cyclic terms are not supported because this would require establishing a canonical cycle. That is, given A=[a|A] and B=[a,a|B], *A* and *B* should produce the same hash. This is not (yet) implemented.

This hash was developed for lookup of solutions to a goal stored in a table. By using a cryptographic hash, heuristic algorithms can often ignore the possibility of hash collisions and thus avoid storing the goal term itself as well as testing using [=@=/2](http://www.swi-prolog.org/pldoc/man?predicate=%3D@%3D/2).

**variant\_hash**(*+Term, -HashKey*)

Similar to [variant\_sha1/2](http://www.swi-prolog.org/pldoc/man?predicate=variant_sha1/2), but using a non-cryptographic hash and produces an integer result like [term\_hash/2](http://www.swi-prolog.org/pldoc/man?predicate=term_hash/2). This version does deal with attributed variables, processing them as normal variables. This hash is primarily intended to speedup finding variant terms in a set of terms.bug

**version**

Write the SWI-Prolog banner message as well as additional messages registered using[version/1](http://www.swi-prolog.org/pldoc/man?predicate=version/1). This is the default *initialization goal* which can be modified using **-g**.

**version**(*+Message*)

Register additional messages to be printed by [version/0](http://www.swi-prolog.org/pldoc/man?predicate=version/0). Each registered message is handed to the message translation DCG and can thus be defined using the hook prolog:message//1. If not defined, it is simply printed.

**visible**(*+Ports*)

Set the ports shown by the debugger. See [leash/1](http://www.swi-prolog.org/pldoc/man?predicate=leash/1) for a description of the *Ports* specification. Default is full.

**volatile** *+Name/Arity, ...*

Declare that the clauses of specified predicates should **not** be saved to the program. The volatile declaration is normally used to prevent the clauses of dynamic predicates that represent data for the current session from being saved in the state file.

**wait\_for\_input**(*+ListOfStreams, -ReadyList, +TimeOut*)

Wait for input on one of the streams in *ListOfStreams* and return a list of streams on which input is available in *ReadyList*. [wait\_for\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=wait_for_input/3) waits for at most *TimeOut* seconds. *Timeout*may be specified as a floating point number to specify fractions of a second. If *Timeout* equalsinfinite, [wait\_for\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=wait_for_input/3) waits indefinitely.

This predicate can be used to implement timeout while reading and to handle input from multiple sources. The following example will wait for input from the user and an explicitly opened second terminal. On return, *Inputs* may hold user\_input or *P4* or both.

?- open('/dev/ttyp4', read, P4),

wait\_for\_input([user\_input, P4], Inputs, 0).

This predicate relies on the select() call on most operating systems. On Unix this call is implemented for any stream referring to a file handle, which implies all OS-based streams: sockets, terminals, pipes, etc. On non-Unix systems select() is generally only implemented for socket-based streams. See also library(socket) from the clib package.

Note that [wait\_for\_input/3](http://www.swi-prolog.org/pldoc/man?predicate=wait_for_input/3) returns streams that have data waiting. This does not mean you can, for example, call [read/2](http://www.swi-prolog.org/pldoc/man?predicate=read/2) on the stream without blocking as the stream might hold an incomplete term. The predicate [set\_stream/2](http://www.swi-prolog.org/pldoc/man?predicate=set_stream/2) using the option timeout(Seconds) can be used to make the stream generate an exception if no new data arrives within the timeout period. Suppose two processes communicate by exchanging Prolog terms. The following code makes the server immune for clients that write an incomplete term:

...,

tcp\_accept(Server, Socket, \_Peer),

tcp\_open(Socket, In, Out),

set\_stream(In, timeout(10)),

catch(read(In, Term), \_, (close(Out), close(In), fail)),

...,

**when**(*@Condition, :Goal*)

Execute *Goal* when *Condition* becomes true. *Condition* is one of ?=(X, Y), nonvar(X),ground(X), ,(Cond1, Cond2) or ;(Cond1, Cond2). See also [freeze/2](http://www.swi-prolog.org/pldoc/man?predicate=freeze/2) and [dif/2](http://www.swi-prolog.org/pldoc/man?predicate=dif/2). The implementation can deal with cyclic terms in *X* and *Y*.

The [when/2](http://www.swi-prolog.org/pldoc/man?predicate=when/2) predicate is realised using attributed variables associated with the module when. It is defined in the autoload library library(when).

**wildcard\_match**(*+Pattern, +String*)

True if *String* matches the wildcard pattern *Pattern*. *Pattern* is very similar to the Unix cshpattern matcher. The patterns are given below:

|  |  |
| --- | --- |
| ? | Matches one arbitrary character. |
| \* | Matches any number of arbitrary characters. |
| [ ... ] | Matches one of the characters specified between the brackets. |
|  | <*char1*>-<*char2*> indicates a range. |
| {...} | Matches any of the patterns of the comma-separated list between the braces. |

Example:

?- wildcard\_match('[a-z]\*.{pro,pl}[%~]', 'a\_hello.pl%').

true.

**win\_add\_dll\_directory**(*+AbsDir*)

This predicate adds a directory to the search path for dependent DLL files. If possible, this is achieved with [win\_add\_dll\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=win_add_dll_directory/2). Otherwise, %PATH% is extended with the provided directory. *AbsDir* may be specified in the Prolog canonical syntax. See[prolog\_to\_os\_filename/2](http://www.swi-prolog.org/pldoc/man?predicate=prolog_to_os_filename/2). Note that [use\_foreign\_library/1](http://www.swi-prolog.org/pldoc/man?predicate=use_foreign_library/1) passes an absolute path to the DLL if the destination DLL can be located from the specification using [absolute\_file\_name/3](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/3).

**win\_add\_dll\_directory**(*+AbsDir, -Cookie*)

This predicate adds a directory to the search path for dependent DLL files. If the call is successful it unifies *Cookie* with a handle that must be passed to [win\_remove\_dll\_directory/1](http://www.swi-prolog.org/pldoc/man?predicate=win_remove_dll_directory/1)to remove the directory from the search path. Error conditions:

* This predicate is available in the Windows port of SWI-Prolog starting from 6.3.8/6.2.6.
* This predicate *fails* if Windows does not yet support the underlying primitives. These are available in recently patched Windows 7 systems and later.
* This predicate throws an acception if the provided path is invalid or the underlying Windows API returns an error.

If [open\_shared\_object/2](http://www.swi-prolog.org/pldoc/man?predicate=open_shared_object/2) is passed an *absolute* path to a DLL on a Windows installation that supports AddDllDirectory() and friends,118 SWI-Prolog uses LoadLibraryEx() with the flagsLOAD\_LIBRARY\_SEARCH\_DLL\_LOAD\_DIR and LOAD\_LIBRARY\_SEARCH\_DEFAULT\_DIRS. In this scenario, directories from %PATH% and *not* searched. Additional directories can be added using [win\_add\_dll\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=win_add_dll_directory/2).

**win\_remove\_dll\_directory**(*-Cookie*)

Remove a DLL search directory installed using [win\_add\_dll\_directory/2](http://www.swi-prolog.org/pldoc/man?predicate=win_add_dll_directory/2).

**win\_exec**(*+Command, +Show*)

Windows only. Spawns a Windows task without waiting for its completion. *Show* is one of the Win32 SW\_\* constants written in lowercase without the SW\_\*: hide maximize minimizerestore show showdefault showmaximized showminimized showminnoactive shownashownoactive shownormal. In addition, iconic is a synonym for minimize and normal forshownormal.

**win\_exec**(*+Command, +Show*)

Windows only. Spawns a Windows task without waiting for its completion. *Show* is one of the Win32 SW\_\* constants written in lowercase without the SW\_\*: hide maximize minimizerestore show showdefault showmaximized showminimized showminnoactive shownashownoactive shownormal. In addition, iconic is a synonym for minimize and normal forshownormal.

**win\_has\_menu**

True if [win\_insert\_menu/2](http://www.swi-prolog.org/pldoc/man?predicate=win_insert_menu/2) and [win\_insert\_menu\_item/4](http://www.swi-prolog.org/pldoc/man?predicate=win_insert_menu_item/4) are present.

**win\_folder**(*?Name, -Directory*)

True if *Name* is the Windows `CSIDL' of *Directory*. If *Name* is unbound, all known Windows special paths are generated. *Name* is the CSIDL after deleting the leading CSIDL\_ and mapping the constant to lowercase. Check the Windows documentation for the function SHGetSpecialFolderPath() for a description of the defined constants. This example extracts the `My Documents' folder:

?- win\_folder(personal, MyDocuments).

MyDocuments = 'C:/Documents and Settings/jan/My Documents'

**win\_insert\_menu**(*+Label, +Before*)

Insert a new entry (pulldown) in the menu. If the menu already contains this entry, nothing is done. The *Label* is the label and, using the Windows convention, a letter prefixed with & is underlined and defines the associated accelerator key. *Before* is the label before which this one must be inserted. Using - adds the new entry at the end (right). For example, the call below adds an **Application** entry just before the **Help** menu.

win\_insert\_menu('&Application', '&Help')

**win\_insert\_menu\_item**(*+Pulldown, +Label, +Before, :Goal*)

Add an item to the named *Pulldown* menu. *Label* and *Before* are handled as in[win\_insert\_menu/2](http://www.swi-prolog.org/pldoc/man?predicate=win_insert_menu/2), but the label - inserts a *separator*. *Goal* is called if the user selects the item.

**win\_shell**(*+Operation, +File, +Show*)

Windows only. Opens the document *File* using the Windows shell rules for doing so.*Operation* is one of open, print or explore or another operation registered with the shell for the given document type. On modern systems it is also possible to pass a URL as *File*, opening the URL in Windows default browser. This call interfaces to the Win32 API ShellExecute(). The*Show* argument determines the initial state of the opened window (if any). See [win\_exec/2](http://www.swi-prolog.org/pldoc/man?predicate=win_exec/2) for defined values.

**win\_shell**(*+Operation, +File*)

Same as win\_shell(Operation, File, normal)

**win\_registry\_get\_value**(*+Key, +Name, -Value*)

Windows only. Fetches the value of a Windows registry key. *Key* is an atom formed as a path name describing the desired registry key. *Name* is the desired attribute name of the key. *Value*is unified with the value. If the value is of type DWORD, the value is returned as an integer. If the value is a string, it is returned as a Prolog atom. Other types are currently not supported. The default `root' is HKEY\_CURRENT\_USER. Other roots can be specified explicitly asHKEY\_CLASSES\_ROOT, HKEY\_CURRENT\_USER, HKEY\_LOCAL\_MACHINE or HKEY\_USERS. The example below fetches the extension to use for Prolog files (see README.TXT on the Windows version):

?- win\_registry\_get\_value(

'HKEY\_LOCAL\_MACHINE/Software/SWI/Prolog',

fileExtension,

Ext).

Ext = pl

**win\_window\_pos**(*+ListOfOptions*)

Interface to the MS-Windows SetWindowPos() function, controlling size, position and stacking order of the window. *ListOfOptions* is a list that may hold any number of the terms below:

**size**(*W, H*)

Change the size of the window. *W* and *H* are expressed in character units.

**position**(*X, Y*)

Change the top-left corner of the window. The values are expressed in pixel units.

**zorder**(*ZOrder*)

Change the location in the window stacking order. Values are bottom, top, topmost andnotopmost. *Topmost* windows are displayed above all other windows.

**show**(*Bool*)

If true, show the window, if false hide the window.

**activate**

If present, activate the window.

**window\_title**(*-Old, +New*)

Unify *Old* with the title displayed in the console and change the title to *New*.bug

**with\_mutex**(*+MutexId, :Goal*)

Execute *Goal* while holding *MutexId*. If *Goal* leaves choice points, these are destroyed (as in[once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1)). The mutex is unlocked regardless of whether *Goal* succeeds, fails or raises an exception. An exception thrown by *Goal* is re-thrown after the mutex has been successfully unlocked. See also [mutex\_create/1](http://www.swi-prolog.org/pldoc/man?predicate=mutex_create/1) and [setup\_call\_cleanup/3](http://www.swi-prolog.org/pldoc/man?predicate=setup_call_cleanup/3).

Although described in the thread section, this predicate is also available in the single-threaded version, where it behaves simply as [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1).

**with\_output\_to**(*+Output, :Goal*)

Run *Goal* as [once/1](http://www.swi-prolog.org/pldoc/man?predicate=once/1), while characters written to the current output are sent to *Output*. The predicate is SWI-Prolog-specific, inspired by various posts to the mailinglist. It provides a flexible replacement for predicates such as sformat/3 , [swritef/3](http://www.swi-prolog.org/pldoc/man?predicate=swritef/3), [term\_to\_atom/2](http://www.swi-prolog.org/pldoc/man?predicate=term_to_atom/2),[atom\_number/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_number/2) converting numbers to atoms, etc. The predicate [format/3](http://www.swi-prolog.org/pldoc/man?predicate=format/3) accepts the same terms as output argument.

Applications should generally avoid creating atoms by breaking and concatenating other atoms, as the creation of large numbers of intermediate atoms generally leads to poor performance, even more so in multithreaded applications. This predicate supports creating difference lists from character data efficiently. The example below defines the DCG ruleterm/3 to insert a term in the output:

term(Term, In, Tail) :-

with\_output\_to(codes(In, Tail), write(Term)).

?- phrase(term(hello), X).

X = [104, 101, 108, 108, 111]

**A Stream handle or alias**

Temporarily switch current output to the given stream. Redirection using[with\_output\_to/2](http://www.swi-prolog.org/pldoc/man?predicate=with_output_to/2) guarantees the original output is restored, also if *Goal* fails or raises an exception. See also [call\_cleanup/2](http://www.swi-prolog.org/pldoc/man?predicate=call_cleanup/2).

**atom**(*-Atom*)

Create an atom from the emitted characters. Please note the remark above.

**string**(*-String*)

Create a string object as defined in [section 5.2](http://www.swi-prolog.org/pldoc/man?section=strings).

**codes**(*-Codes*)

Create a list of character codes from the emitted characters, similar to [atom\_codes/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_codes/2).

**codes**(*-Codes, -Tail*)

Create a list of character codes as a difference list.

**chars**(*-Chars*)

Create a list of one-character atoms from the emitted characters, similar to[atom\_chars/2](http://www.swi-prolog.org/pldoc/man?predicate=atom_chars/2).

**chars**(*-Chars, -Tail*)

Create a list of one-character atoms as a difference list.

**with\_quasi\_quotation\_input**(*+Content, -Stream, :Goal*)

Process the quasi-quoted *Content* using *Stream* parsed by *Goal*. *Stream* is a temporary stream with the following properties:

* Its initial *position* represents the position of the start of the quoted material.
* It is a text stream, using utf8 *encoding*.
* It allows for repositioning
* It will be closed after *Goal* completes.

|  |  |
| --- | --- |
| ***Goal*** | is executed as once(Goal). *Goal* must succeed. Failure or exceptions from *Goal* are interpreted as syntax errors. |

**See also**

[phrase\_from\_quasi\_quotation/2](http://www.swi-prolog.org/pldoc/man?predicate=phrase_from_quasi_quotation/2) can be used to process a quotation using a grammar.

**working\_directory**(*-Old, +New*)

Unify *Old* with an absolute path to the current working directory and change working directory to *New*. Use the pattern working\_directory(CWD, CWD) to get the current directory. See also [absolute\_file\_name/2](http://www.swi-prolog.org/pldoc/man?predicate=absolute_file_name/2) and [chdir/1](http://www.swi-prolog.org/pldoc/man?predicate=chdir/1).bug Note that the working directory is shared between all threads.

**write**(*+Term*)

Write *Term* to the current output, using brackets and operators where appropriate.

**write**(*+Stream, +Term*)

Write *Term* to *Stream*.

**writeln**(*+Term*)

Equivalent to write(Term), nl.. The output stream is locked, which implies no output from other threads can appear between the term and newline.

**writeln**(*+Stream, +Term*)

Equivalent to write(Stream, Term), nl(Stream).. The output stream is locked, which implies no output from other threads can appear between the term and newline.

**write\_canonical**(*+Term*)

Write *Term* on the current output stream using standard parenthesised prefix notation (i.e., ignoring operator declarations). Atoms that need quotes are quoted. Terms written with this predicate can always be read back, regardless of current operator declarations. Equivalent to[write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) using the options ignore\_ops, quoted and numbervars after [numbervars/4](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/4)using the singletons option.

Note that due to the use of [numbervars/4](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/4), non-ground terms must be written using a *single*[write\_canonical/1](http://www.swi-prolog.org/pldoc/man?predicate=write_canonical/1) call. This used to be the case anyhow, as garbage collection between multiple calls to one of the write predicates can change the \_G<NNN> identity of the variables.

**write\_canonical**(*+Stream, +Term*)

Write *Term* in canonical form on *Stream*.

**write\_length**(*+Term, -Length, +Options*)

True when *Length* is the number of characters emitted for *write\_term*Term, Options . In addition to valid options for [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2), it processes the option:

**max\_length**(*+MaxLength*)

If provided, fail if *Length* would be larger than *MaxLength*. The implementation ensures that the runtime is limited when computing the length of a huge term with a bounded maximum.

**write\_term**(*+Term, +Options*)

The predicate [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) is the generic form of all Prolog term-write predicates. Valid options are:

**attributes**(*Atom*)

Define how attributed variables (see [section 7.1](http://www.swi-prolog.org/pldoc/man?section=attvar)) are written. The default is determined by the Prolog flag [write\_attributes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:write_attributes). Defined values are ignore (ignore the attribute),dots (write the attributes as {...}), write (simply hand the attributes recursively to[write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2)) and portray (hand the attributes to [attr\_portray\_hook/2](http://www.swi-prolog.org/pldoc/man?predicate=attr_portray_hook/2)).

**back\_quotes**(*Atom*)

Fulfills the same role as the [back\_quotes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:back_quotes) prolog flag. Notably, the value string causes string objects to be printed between back quotes and symbol\_char causes the backquote to be printed unquoted. In all other cases the backquote is printed as a quoted atom.

**brace\_terms**(*Bool*)

If true (default), write {}(X) as {X}. See also dotlists and ignore\_ops.

**blobs**(*Atom*)

Define how non-text blobs are handled. By default, this is left to the write handler specified with the blob type. Using portray, [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) is called for each blob encountered. See [section 10.4.7](http://www.swi-prolog.org/pldoc/man?section=foreigninclude).

**character\_escapes**(*Bool*)

If true and quoted(true) is active, special characters in quoted atoms and strings are emitted as ISO escape sequences. Default is taken from the reference module (see below).

**cycles**(*Bool*)

If true (default), cyclic terms are written as @(Template, Substitutions), where*Substitutions* is a list *Var* = *Value*. If cycles is false, max\_depth is not given, and *Term* is cyclic, [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) raises a domain\_error.81 See also the cycles option in [read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2).

**dotlists**(*Bool*)

If true (default false), write lists using the dotted term notation rather than the list notation.82 Note that as of version 7, the list constructor is '[|]'. Usingdotlists(true), [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2) writes a list using `.' as constructor. This is intended for communication with programs such as other Prolog systems, that rely on this notation.

**fullstop**(*Bool*)

If true (default false), add a fullstop token to the output. The dot is preceeded by a space if needed and followed by a space (default) or newline if the nl(true) option is also given.83

**ignore\_ops**(*Bool*)

If true, the generic term representation (<*functor*>(<*args*> ... )) will be used for all terms. Otherwise (default), operators will be used where appropriate.84.

**max\_depth**(*Integer*)

If the term is nested deeper than *Integer*, print the remainder as ellipses ( ... ). A 0 (zero) value (default) imposes no depth limit. This option also delimits the number of printed items in a list. Example:

?- write\_term(a(s(s(s(s(0)))), [a,b,c,d,e,f]),

[max\_depth(3)]).

a(s(s(...)), [a, b|...])

true.

Used by the top level and debugger to limit screen output. See also the Prolog flags[answer\_write\_options](http://www.swi-prolog.org/pldoc/man?section=flags#flag:answer_write_options) and [debugger\_write\_options](http://www.swi-prolog.org/pldoc/man?section=flags#flag:debugger_write_options).

**module**(*Module*)

Define the reference module (default user). This defines the default value for the[character\_escapes](http://www.swi-prolog.org/pldoc/man?section=flags#flag:character_escapes) option as well as the operator definitions to use. See also [op/3](http://www.swi-prolog.org/pldoc/man?predicate=op/3).

**nl**(*Bool*)

Add a newline to the output. See also the fullstop option.

**numbervars**(*Bool*)

If true, terms of the format $VAR(N), where *N* is a non-negative integer, will be written as a variable name. If *N* is an atom it is written without quotes. This extension allows for writing variables with user-provided names. The default is false. See also[numbervars/3](http://www.swi-prolog.org/pldoc/man?predicate=numbervars/3) and the option variable\_names.

**partial**(*Bool*)

If true (default false), do not reset the logic that inserts extra spaces that separate tokens where needed. This is intended to solve the problems with the code below. Calling write\_value(.) writes .., which cannot be read. By adding partial(true) to the option list, it correctly emits . .. Similar problems appear when emitting operators using multiple calls to [write\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=write_term/3).

write\_value(Value) :-

write\_term(Value, [partial(true)]),

write('.'), nl.

**portray**(*Bool*)

Same as portrayed(Bool). Deprecated.

**portray\_goal**(*:Goal*)

Implies portray(true), but calls *Goal* rather than the predefined hook [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1). *Goal*is called through call/3, where the first argument is *Goal*, the second is the term to be printed and the 3rd argument is the current write option list. The write option list is copied from the write\_term call, but the list is guaranteed to hold an option prioritythat reflects the current priority.

**portrayed**(*Bool*)

If true, the hook [portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) is called before printing a term that is not a variable. If[portray/1](http://www.swi-prolog.org/pldoc/man?predicate=portray/1) succeeds, the term is considered printed. See also [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1). The default isfalse. This option is an extension to the ISO write\_term options.

**priority**(*Integer*)

An integer between 0 and 1200 representing the `context priority'. Default is 1200. Can be used to write partial terms appearing as the argument to an operator. For example:

format('~w = ', [VarName]),

write\_term(Value, [quoted(true), priority(699)])

**quoted**(*Bool*)

If true, atoms and functors that need quotes will be quoted. The default is false.

**spacing**(*+Spacing*)

Determines whether and where extra white space is added to enhance readability. The default is standard, adding only space where needed for proper tokenization by[read\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=read_term/3). Currently, the only other value is next\_argument, adding a space after a comma used to separate arguments in a term or list.

**variable\_names**(*+List*)

Assign names to variables in *Term*. *List* is a list of terms *Name* = *Var*, where *Name* is an atom that represents a valid Prolog variable name. Terms where *Var* is bound or is a variable that does not appear in *Term* are ignored. Raises an error if *List* is not a list, one of the members is not a term *Name* = *Var*, *Name* is not an atom or *Name* does not represent a valid Prolog variable name.

The implementation binds the variables from *List* to a term '$VAR'(*Name*). Like[write\_canonical/1](http://www.swi-prolog.org/pldoc/man?predicate=write_canonical/1), terms that where already bound to '$VAR'(*X*) before [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2)are printed normally, unless the option numbervars(true) is also provided. If the option numbervars(true) is used, the user is responsible for avoiding collisions between assigned names and numbered names. See also the variable\_names option of[read\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=read_term/2).

Possible variable attributes (see [section 7](http://www.swi-prolog.org/pldoc/man?section=extvar)) are ignored. In most cases one should use[copy\_term/3](http://www.swi-prolog.org/pldoc/man?predicate=copy_term/3) to obtain a copy that is free of attributed variables and handle the associated constraints as appropriate for the use-case.

**write\_term**(*+Stream, +Term, +Options*)

As [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2), but output is sent to *Stream* rather than the current output.

**writef**(*+Atom*)

Equivalent to writef(Atom, []). See [writef/2](http://www.swi-prolog.org/pldoc/man?predicate=writef/2) for details.

**writef**(*+Format, +Arguments*)

Formatted write. *Format* is an atom whose characters will be printed. *Format* may contain certain special character sequences which specify certain formatting and substitution actions.*Arguments* provides all the terms required to be output.

Escape sequences to generate a single special character:

|  |  |
| --- | --- |
| \n | Output a newline character (see also nl/[0,1]) |
| \l | Output a line separator (same as \n) |
| \r | Output a carriage return character (ASCII 13) |
| \t | Output the ASCII character TAB (9) |
| \\ | The character \ is output |
| \% | The character % is output |
| \nnn | where <*nnn*> is an integer (1-3 digits); the character with code <*nnn*> is output (NB : <*nnn*> is read as **decimal**) |

Note that \l, \nnn and \\ are interpreted differently when character escapes are in effect. See[section 2.15.2.1](http://www.swi-prolog.org/pldoc/man?section=syntax).

Escape sequences to include arguments from *Arguments*. Each time a % escape sequence is found in *Format* the next argument from *Arguments* is formatted according to the specification.

|  |  |
| --- | --- |
| %t | [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) the next item (mnemonic: term) |
| %w | [write/1](http://www.swi-prolog.org/pldoc/man?predicate=write/1) the next item |
| %q | [writeq/1](http://www.swi-prolog.org/pldoc/man?predicate=writeq/1) the next item |
| %d | Write the term, ignoring operators. See also [write\_term/2](http://www.swi-prolog.org/pldoc/man?predicate=write_term/2). Mnemonic: old Edinburgh [display/1](http://www.swi-prolog.org/pldoc/man?predicate=display/1) |
| %p | [print/1](http://www.swi-prolog.org/pldoc/man?predicate=print/1) the next item (identical to %t) |
| %n | Put the next item as a character (i.e., it is a character code) |
| %r | Write the next item N times where N is the second item (an integer) |
| %s | Write the next item as a String (so it must be a list of characters) |
| %f | Perform a [ttyflush/0](http://www.swi-prolog.org/pldoc/man?predicate=ttyflush/0) (no items used) |
| %Nc | Write the next item Centered in *N* columns |
| %Nl | Write the next item Left justified in *N* columns |
| %Nr | Write the next item Right justified in *N* columns. *N* is a decimal number with at least one digit. The item must be an atom, integer, float or string. |

**writeq**(*+Term*)

Write *Term* to the current output, using brackets and operators where appropriate. Atoms that need quotes are quoted. Terms written with this predicate can be read back with [read/1](http://www.swi-prolog.org/pldoc/man?predicate=read/1)provided the currently active operator declarations are identical.

**writeq**(*+Stream, +Term*)

Write *Term* to *Stream*, inserting quotes.

**Приложение В**

**Пример отчета №1 на языке Пролог**

**Цель работы:**

1. Ознакомиться с принципами логического программирования на языке Турбо-Пролог.

2. Научиться составлять программы на языке Турбо-Пролог.

**Задание:**

Средствами языка Prolog создать программу, которая решает кроссворд путем подбора слов из базы данных.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | *О* |  |  | *У* |  |  |
|  | *С* |  |  | *Ш* |  |  |
| *Т* | *А* | *Й* | *Ф* | *У* | *Н* |  |
|  |  |  | *О* |  |  |  |
|  |  |  | *Н* |  |  |  |

**СЛОВА***: ТАЙФУН, ОСА, ОБА, УШУ, ФОН, ФЕЯ.* Найти и вывести в отдельном окне все возможные варианты. Каждый новый вариант должен выводиться при нажатии любой клавиши. Один из возможных вариантов приведен на рисунке

**Код программы**

domains

w = symbol

predicates

wr(w)

Window

solve

findWord(integer, w)

getChar(integer, w, w)

findWordWithChar(integer, w, integer, w)

goal

Window,

solve.

clauses

wr("tayfun").

wr("osa").

wr("oba").

wr("usu").

wr("fon").

wr("fey").

createWindow:-

makeWindow(1,123,4,"LABA1\_Kalinin",4,5,10,40).

findWord(LEN,WORD):-

wr(WORD),

str\_len(WORD,LEN).

getChar(POSITION,WORD,CH):-

frontstr(POSITION,WORD,X,Y),

str\_len(X,LX),

LNX=LX-1,

frontstr(LNX,X,A,CH).

findWordWithChar(LEN,CH,POSITION,WORD):-

findWord(LEN,WORD),

getChar(POSITION,WORD,NCH),

NCH=CH.

solve:-

findWord(6,W),

getChar(2,W,C),

findWordWithChar(3,C,3,W1),

getChar(4,W,C1),

findWordWithChar(3,C1,1,W2),

getChar(5,W,C2),

findWordWithChar(3,C2,3,W3),

W2<>W3,

W1<>W2,

getChar(1,W1,CW01),

getChar(2,W1,CW02),

getChar(2,W2,CW03),

getChar(3,W2,CW04),

getChar(1,W3,CW05),

getChar(2,W3,CW06),

clearWindow,

write(" ",CW01," "," ",CW05," "),nl,

write(" ",CW02," "," ",CW06," "),nl,

write(W),nl,

write(" "," "," ",CW03," "," "),nl,

write(" "," "," ",CW04," "," "),nl,

readChar(RCH),

fail.

**Результат работы программы:**

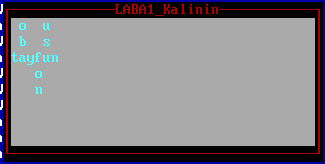


Рисунок 1 – Результат работы программы

**Выводы:**

В ходе лабораторной работы были изучены принципы логического программирования на языке Турбо-Пролог. Составлена программа, для решения кроссворда.

**Пример отчета №2 на языке Пролог**

#### Цель работы

В результате выполнения лабораторной работы необходимо:

* ознакомиться с принципами логического программирования на языке Пролог;
* научиться составлять программы на языке Пролог.

1. Задание на лабораторную работу

Вариант №6

Средствами языка Prolog создать программу, которая решает логическую задачу.

Барсук позвал к себе гостей:  
Медведя, рысь и белку.  
И подарили барсуку  
Подсвечник и тарелку.  
Когда же он позвал к себе Рысь, белку, мышку, волка,  
о он в подарок получил подсвечник и иголку.  
Им были вновь приглашены  
волк, мышка и овечка.  
И получил в подарок он  
Иголку и колечко.  
Он снова пригласил овцу,  
Медведя, Волка белку  
Колечко и тарелку.  
Нам срочно нужен ваш совет.  
(На миг дела отбросьте.)  
Хотим понять, Какой предмет   
Каким подарен гостем,  
И кто из шестерых гостей  
Явился без подарка?  
Не можем мы сообразить,  
Сидим... Мудрим... Запарка...

1. Решение задачи табличным способом

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Медведь | Рысь | Овца | Белка | Волк | Мышь |
| Кольцо | - | - | + | - | - | - |
| Подсвечник | - | + | - | - | - | - |
| Тарелка | + | - | - | - | - | - |
| Иголка | - | - | - | - | - | + |
| Ничего | - | - | - | + | + | - |

Ответ: Волк и Белка ничего не подарили, мышь – иголку, медведь – тарелку, овца – кольцо, рысь - подсвечник.

1. Текст программы:

ame(bobcat).

name(squirrel).

name(mouse).

name(wolf).

name(sheep).

name(bear).

gift(ring).

gift(plate).

gift(candlestick).

gift(needle).

gift(nothing).

ngive(bobcat,ring).

ngive(bobcat,needle).

ngive(bear,ring).

ngive(bear,needle).

ngive(squirrel,ring).

ngive(squirrel,needle).

ngive(wolf,plate).

ngive(wolf,ring).

ngive(bobcat,plate).

ngive(mouse,plate).

ngive(mouse,ring).

ngive(squirrel,plate).

ngive(sheep,candlestick).

ngive(sheep,plate).

ngive(mouse,candlestick).

ngive(wolf,candlestick).

ngive(bear,candlestick).

ngive(sheep,needle).

ngive(bear,candlestick).

ngive(wolf,needle).

ngive(squirrel,candlestick).

rez(X,Y):- name(X),gift(Y),not(ngive(X,Y)).

output(X1,Y1,X2,Y2,X3,Y3,X4,Y4,X5,Y5,X6,Y6):- X1=bear,rez(X1,Y1),X2=mouse,rez(X2,Y2),X3=squirrel,

rez(X3,Y3),X4=wolf,rez(X4,Y4),X5=bobcat,rez(X5,Y5),X6=sheep,rez(X6,Y6). name(bobcat).

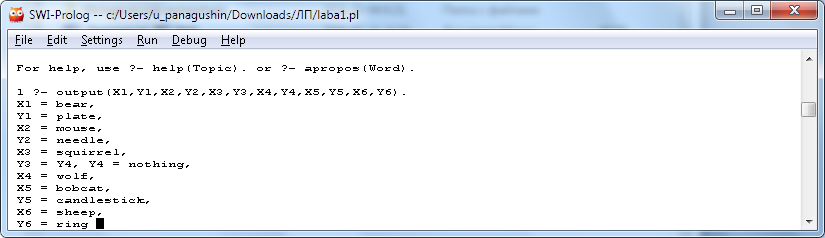


Рис.1 – Результата работы

***Выводы:***

В результате выполнения лабораторной работы получены навыки написания программ на языке Prolog. В качестве диалекта пролога использовался SWI Prolog (GNU Prolog) версия для Unix систем в адаптированном под Windows варианте. Среди существенных отличий данного диалекта от Турбо-пролога нужно отметить отсутствие деления на секции (DOMAINS, PREDICATES, GOAL и т.д.) и ряд отличий в синтаксисе, например таких, как замена операции отрицания равенства с <> на \=.

Данная задача была решена без использования списков, входные посылки писались в качестве фактов в порядке их следования. Для решения задачи был выбран метод “от противного”, исходные посылки переворачивались на изнанку (факт ngive) и содержали сведения о том, что животные точно не дарили Барсуку. Результирующий предикат REZ делает подстановку X и Y в отрицание ngive. Так же была добавлена исходная посылка gift(nothing), которая присваивается тем животным, которые не дарили подарки.

Программа, написанная на прологе, позволяет решать подобные логические задачи без алгоритма, простым перебором посылок. Это существенно может сэкономить время при разработке решения, когда набор входных данных составляет сотни и тысячи посылок.