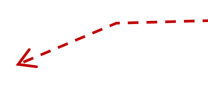
Chapter 6.
Predefined Predicates



Reorganized entire chapter

Updated: 12 June, 2017

Input and output

Exception handling

Type Checking

Comparing

Arithmetic

Files – Naming and Locating

- File names use Unix notation ('/' as separator)
  - … also on Windows!
  - To use Windows notation each '\' must be doubled, because \ is the escape character in Prolog atoms! (e.g. \t represents a tab).
- Aliases
  - Aliases are defined via file_search_path(+Alias, +Path)
  - A file location relative to an alias is specified using the alias as a functor and adding the relative path as argument

```
?- file_search_path(demo, '/home/gk/prolog/demo').

?- absolute_file_name(demo(myfile), Absolute).
Absolute = '/home/gk/prolog/demo/myfile'
true.
```

  - file_search_path/2 is used by absolute_file_name/[2,3] and all loading predicates
  - ?- prolog_flag(verbose_file_search, true) can be used to debug Prolog's search for files.

Prolog Files – Compilation and Loading

- consult(+File)
  - Compile and load prolog files

- make/0
  - Check which loaded files have been modified and reconsult them

Implicit versus Explicit I/O
"Three ways to happiness"

Explicit "ISO style"	Implicit "Edingburg style"	Implicit "SWI style"
• File opening creates a stream identifier ◆ open/3 • All other file operations take a Stream identifier as parameter ◆ read/2, write/2	• There is one global input and output ◆ seeing/1, telling/1 • It is globally redirected ◆ see/1, tell/1 ◆ seen/1, told/1	• Output of a predicate redirected at call site ◆ with_output_to/2 • Special redirection to ◆ atoms ◆ strings

universitätbonn

Streams – I/O with Explicit Src/Dest (1)

open(+*File*, +*Mode*, ?*Stream*) – open a stream

- *File* = *file specifier* | ‘pipe(*Command*)’
  - A file specifier is a path or an alias.
- *Mode* = *read, write, append* or *update*.
  - Mode append positions the file-pointer at the end.
  - Mode update positions the file-pointer at the beginning of the file without truncating it.
- *Stream* is
  - 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.

© 2009 -2017 Dr. G. Kniesel

Streams – I/O with Explicit Src/Dest (2)

current_stream(*?File, ?Mode, +Stream)* – get the status of a stream

current_stream(–*File, ?Mode, –Stream)* – enumerate all open streams

- *Mode = read | write*

set stream position(+*Stream*, +*Pos)* – Set current position of *Stream to Pos.*

Many more stream handling predicates (see manual)!

Explicit versus Implicit I/O “Three ways to happiness”

Explicit “ISO style”	Implicit “Edingburg style”	Implicit “SWI style”
- File opening creates a stream identifier - open/3 - All other file operations take a Stream identifier as parameter - read/2, write/2	- There is one global input and output - seeing/1, telling/1 - It is globally redirected - see/1, tell/1 - seen/1, told/1	
- Bad - pass stream to all predicates that might need I/O … or assert stream information	- Good - simple idiom for input switching - Bad - global state - interference possible	

Files – I/O with implicit Src/Dest (1)

The reading predicates refer to the implicit current input stream
The writing predicates refer to the implicit current output stream.

- Initially both are connected to the terminal / console.

- The current input stream is
  - obtained using `seeing/1`.
  - changed using `see/1`.
  - closed using `seen/0`.
- The current output stream is
  - obtained using `telling/1`
  - changed using `tell/1` or `append/1`.
  - closed using `told/0`.

Write at the end of the file

- The arguments of these operations are either
  - a file
  - `user` (the reserved stream name for the terminal)
  - `pipe(`*Command*`)`

```
?- see(data).        % Start reading from file 'data'.
?- tell(user).       % Start writing to the terminal.
?- tell(pipe(lpr)).  % Start writing to the printer.
```


universitätbonn

Using Implicit Source

- Application example: Consulting a file

```
consult(File) :-
    seeing(OldFile), see(File),    % set input to File
    consult_loop,                  % read File and assert clauses
    seen, see(OldFile).            % reset input to OldFile

consult_loop :-
    repeat,                        <----- succeeds infinitely
       read(Clause),
       process(Clause),
    !.

process(X) :-
    X == end_of_file.              <----- successful end

process(Clause) :-
    assert(Clause),
    fail.                          <----- trigger backtracking
```

Using Implicit Source and Destination

- Application example: Copying a file

```
 copy(Input,Output) :-
     seeing(OldInput),see(Input),        % set new input
     telling(OldOutput),tell(Output),    % set new output
      copy,                              % do the real copying
     seen, see(OldInput),                % reset old input
     told, tell(OldOutput).              % reset old output

copy :-
     repeat,
        read(X),
        write(X),
     X == end_of_file,
     !.
```

- Recall that the idiom used in `copy/0` corresponds to a „do_until“ or „do_whilenot“ loop in imperative languages
  - See section on iteration via backtracking

Implicit versus Explicit I/O "Three ways to happiness"

Explicit "ISO style"	Implicit "Edingburg style"	Implicit "SWI style"
● File opening creates a stream identifier ◆ open/3 ● All other file operations take a Stream identifier as parameter ◆ read/2, write/2 ➢ Bad ◆ pass stream to all predicates that might need I/O or assert stream information	● There is one global input and output ◆ seeing/1, telling/1 ● It is globally redirected ◆ see/1, tell/1 ◆ seen/1, told/1 ➢ Good ◆ simple idiom for input switching ➢ Bad ◆ global state ◆ interference possible	● Output of a predicate redirected at call site ◆ with_output_to/2 ● Special redirection to ◆ atoms ◆ strings ➢ Good ◆ no crosscutting parameter passing ◆ no global state ➢ Bad ◆ with_input_from/2 missing

SWI-Style example

- Asume a predicate that has nothing to do with I/O

```
 ancestor(A,D):- parent(A,D).

copy :-
     repeat,
        read(X),
        write(X),
     X == end_of_file,
     !.
```

- Recall that the idiom used in `copy/0` corresponds to a „do_until“ or „do_whilenot“ loop in imperative languages
  - See section on iteration via backtracking

universitätbonn

Homework (Exam training)

To understand the practical effect of using ISO-style versus Edingburgh-style IO

- Find an example of problematic use of Edingburgh-style IO
- Rewrite the predicates on the previous two slides so that they use ISO-style explicit streams
- Discuss your solution with a colleague who solved the task independently.
- Discuss which style you would prefer and why

Exception Handling

setup_call_cleanup/3

Exception Handling

- Predicate setup_call_cleanup(Setup, Call, Cleanup)
  - First calls Setup
  - Then calls Call
  - Then calls Cleanup

- Cleanup is guaranteed to be executed no matter how Call terminated (success, failure <u>or exception</u>)

Exception Handling: Example 1

- Use setup_call_cleanup(Setup, Call, Cleanup) to close files properly in case of an exception.

```
%% term_in_file(+Term, +File)
% Search Term in File, succeed as soon as Term is found and fail
% if end_of_file is reached before finding it:

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     % if end of file is reached
    -> !, fail              % then fail
    ;  T = Term             % else check if it is the sought term
    ).
```

Exception Handling: Example 2

- Use setup_call_cleanup(Setup, Call, Cleanup) for safe input / output redirection:

```
%% with_input_from( +Stream, :Goal) is semidet.
%
% Switch current input to Stream, call Goal once,
% then switch back to previous input stream.

:- meta_predicate with_input_from(+,0).

with_input_from(Stream,Goal) :-
   current_input(S0),                    % Remember current input
   setup_call_cleanup(set_input(Stream), % Set input to Stream
                      Goal,              % Call Goal
                      set_input(S0)      % Reset the input
   ).
```

Type Checking

Instantiation modes

Term structure

Numbers

Type-checking Built-in Predicates

Testing instantiation modes

- **var(X)**
  - is true when X is an unbound variable.
- **nonvar(X)**
  - is true when X is not a variable or is bound to a variable Y for which nonvar(Y) holds
- **ground**/1
  - argument contains no free variables -- e.g. 1, a, f(X,a) with X bound.

Type-checking Built-in Predicates

Testing term structure

- **atom(X)**
  - X is an atom (but not a number)
- **atomic(X)**
  - **atom(X)** or **number(X).**
- **compound(X)**
  - X is a compound term (a function term)
- **cyclic_term(X)**
  - X is the result of applying a cyclic substituion
- **acyclic_term(X)**
  - not cyclic_term(X)

Testing numbers

- **number(X)**
  - X is an integer or rational
- **rational(X)**
  - rational includes integer
- **integer(X)**
  - X is an integer
- **float(X)**
  - X is a float

Comparing Numbers and Terms

Comparing numbers
Comparing terms

Comparing Numbers

Arithmetic Notation	**Prolog Operator**	
$x < y$	`X < Y`	
$x \leq y$	`X =< Y`	
$x > y$	`X > Y`	
$x \geq y$	`X >= Y`	
$x = y$	`X =:= Y`	← not `X=Y` !!!
$x \neq y$	`X =\= Y`	← not `X\=Y` !!!

All operators that compare numbers

1. evaluate both arguments,
2. cast them appropriately,
3. compare the results

Comparing Numbers: Examples

No surprises

```
?- 2 < 4+1 .
yes

?- 4+3 > 5+5 .
no
```

arithmetic > test

Note the difference!

```
?- 4 = 4 .
yes

?- 2+2 = 4 .
no
```

unification

```
?- 2+2 =:= 4 .
yes

?- 2+2 =:= 4.0 .
yes
```

arithmetic equality test

Comparing Numbers

- Warning: The is/2 predicate should not be used for testing equality
  - Using it with unbound left operand can give unexpected results:

```
?- 1 is sin(pi/2).
false.
```

  - The above fails because `sin(pi/2)` evaluates to the float 1.0, which does not unify with the integer 1

- If equality of numbers is to be tested, =:=/2 should be used

```
?- 1 =:= sin(pi/2).
true.
```

  - The above succeeds as expected because =:=/2 evaluates the right-hand-side expression and casts it appropriately before it tries to unify it to the left-hand-side expression

Comparing Terms

- Operators that compare terms do not evaluate their arguments

```
X @< Y        smaller
X @> Y        greater
X @=< Y       smaller or equal
X @>= Y       greater or equal
X == Y        identical
X \== Y       not identical
```

- They compare them according to the “standard order of terms”
  - Free Variable @< Number @< Atom @< String @< Compound Term
  - Bound variables are sorted like the term to which they are bound
  - Free variables are sorted by address
  - Atoms are sorted alphabetically. Strings too.
  - Numbers are sorted by value.
  - Compound terms are sorted first on their arity, then on their functor and finally on their arguments, leftmost first.

Comparing Terms: Standard Order

- "Standard order of terms"
  - Free Variable @< Number @< Atom @< String @< Compound Term
  - Bound variables are sorted like the term to which they are bound
  - Free variables are sorted by address
  - Atoms are sorted alphabetically. Strings too.
  - Numbers are sorted by value.
  - Compound terms are sorted first on their arity, then on their functor and finally on their arguments, leftmost first.

```
?- _G9669 @< _G9670.   true
?- X @< 1.             true
?- aaa @< b.           true
?- 2 @< 3.             true
?- f(b) @< a(a,a)      true
?- f(b,b) @< g(a,a).   true
?- f(b,b) @< f(c,a).   true
?- f(b,b) @< f(b,c).   true
```

```
?- X == Y.                 false
?- X == X.                 true
?- a == b.                 false
?- a == a.                 true
?- a == A.                 false
?- a=A, a==A.              true
?- X=Y, X==Y.              true
?- X=2, Y=2, X==Y.         true
```

Exercise: Write your own Comparison

- Define a predicate max_elem/2 that is true if:
  - The first argument is a list of positive integers and
  - … the second argument is the highest integer in the list.
  -

```
?- max_elem([1,0,5,4], Max).
Max=5
```

- Solution

```
max_elem(L,Max):- max_elem__(L,0,Max).


max_elem__([],A,A).

max_elem__([H|T],A,Max):-  H > A,  max_elem__(T,H,Max).

max_elem__([H|T],A,Max):-  H =< A, max_elem__(T,A,Max).
```

Arithmetic

is/2 predicate

Defining own arithmetic functions

Arithmetic in Prolog

- Terms are not functions!
  - Expressions such as 3+2, 4-7, 5/5 are ordinary Prolog terms
  - They do not carry out any arithmetic (do not compute a result)!

```
?- X = 3+2.      % just unifies X to 3+2
X = 3+2
yes

?- 3+2 = X.      % just unifies X to 3+2
X = 3+2
yes
```


universitätbonn

Arithmetic in Prolog: The is/2 predicate

- To actually evaluate arithmetic expressions, we must use the **is/2** predicate
- It unifies the left-hand-side term with the result of evaluating the right-hand-side term as an arithmetic expression:

```
?- X is 3+2.
X = 5
true.
```

```
?- 3+2 is 3+2.
false.
```

```
?- 3+2 = 3+2.
true.
```

- Integer Arithmetic
  - Whenever both arguments are integers
  - When a floating point value is whole (→ transformed into an integer).
- Floating point arithmetic
  - if an argument is a non-whole floating point

Arithmetic in Prolog

- Any standard arithmetic operator can be evaluated by “is/2”:

```
?-  X is 2+3.                   % X=5        addition
?-  X is 5-3.                   % X=2        subtraction
?-  X is 3*4.                   % X=12       multiplication
?-  X is 7/2.                   % X=3.5      division
?-  X is round(7/2).            % X=4        rounding
?-  X is 7//2.                  % X=3        integer division
?-  X is mod(7,2).              % X=1        modulo
?-  X is abs(-5-3).             % X=8        absolute value
?-  X is max(2.5,3).            % X=3        maximum
?-  X is float(max(2.5,3)).     % X=3.0      casting to float
?-  X is random(1000).          % 1 <= X <= 1000
```

- SWI-Prolog supports arithmetic, triogonometry, casting, bitvector operations, and many more
  - See SWI-Prolog manual, Section “4.26 Arithemetic”
  - “?- help.” and then search for “arithmetic”

(Un)Safe Invocation Modes

- The is/2 predicate requires the right-hand-side expression to be "ground" (at the time when is/2 is executed):

```
?- 5 is X+2.
ERROR: is/2: Arguments are not sufficiently instantiated
?- X=3, Y is X+2.
X = 3,
Y = 5.
```

- Unsafe invocation modes = Invocation modes for which the predicate cannot be evaluated
  - Typical for many built-in predicates
  - E.g. is/2 with mode (?, nonground) is unsafe

Defining Arithmetic Functions

- arithmetic_function(+Head)
  - Register a new arithmetic function that can be used within the right-hand-side of an is/2 call
  - +Head must be of the form Name/Arity, an atom or a function term.
  - There must exist a predicate that has the name specified by Head and Arity+1 arguments
    - ⇨ The arguments on position 1 to Arity are input parameters that must be bound to numbers at call time.
    - ⇨ The last argument (on position Arity+1) must be unbound at call time and must be bound by the body of the predicate. It will be bound to the left-hand-side of the is/2 call.

```
% Contents of file "mean.pl":
 :- arithmetic_function(mean/2).

 mean(A, B, C) :- C is (A+B)/2.
```

```
?- consult(mean).
mean compiled, 0.07 sec, 440 bytes.

?- A is mean(4,5).
 A = 4.500000
```

Summary

- File I/O
  - File names and aliases
  - Implicit, explicit and SWI-style I/O
- Exception handling
  - setup_call_cleanup/3
- Type checking
  - instantiation
  - term structure
  - numbers
- Comparison
  - numbers
  - terms
- Arithmetic
  - is/2 and own extensions

What‘s next?

- Browse the online documentation at
  http://www.swi-prolog.org/pldoc/doc_for?object=root

- Read about predefined libraries, e.g. socket handling
  http://www.swi-prolog.org/pldoc/doc/swi/library/socket.pl