universitätbonn ROOTS

Chapter 6.
Predefined Predicates

Arithmetic

Testing and comparing

Input and output

Exception handling

Arithmetic

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
```

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 Prolog predicate as an arithmetic function.
  - +Head is either Name/Arity, an atom or a complex term.
  - The predicate must have one more argument than specified by Head.
  - The last argument is an unbound variable at call time and should be instantiated to an integer or floating point number.
  - The other arguments are the parameters.

- Example

```
% 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
```

Comparing Numbers and Terms

Comparing Numbers

- Operators that compare numbers evaluate both arguments and compare the results

Arithmetic	Prolog
$x < y$	`X < Y`
$x \leq y$	`X =< Y`
$x = y$	`X =:= Y`
$x \neq y$	`X =\= Y`
$x \geq y$	`X >= Y`
$x > y$	`X > Y`

Comparing Numbers: Examples

No surprises

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

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

Note the differences!

```
?- 4 = 4 .
yes

?- 2+2 = 4 .
no

?- 2+2 =:= 4 .
yes

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

Comparing Numbers

- Warning: The is/2 predicate should only be used with unbound left operand
  - Using it for testing equality 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 it does not just unify the LHS and RHS but performs comparison including evaluation and conversion

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
```

Special Comparions / Tests (1)

- subsumes(+Generic, +Specific)
  - Generic subsumes Specific if instantiation of Generic produces Specific.
  - Subsumption is also called one-sided unification or semi-unification.

```
?- subsumes(f(X,Y),f(A,c)), writeln((f(X,Y),f(A,c))).

subsumes(f(_G10520, c), f(_G10520, c))
X = A,
Y = c.
```

  - Take care: As a side-effect Generic and Specific will be unified!

- subsumes_chk(+Generic, +Specific)
  - As above but does not unify (only tests)!

```
?- subsumes_chk(f(X,Y),f(A,c)), writeln((f(X,Y),f(A,c))).

subsumes_chk(f(_G9669, _G9670), f(_G9672, c))
true.
```

Special Comparions / Tests (2)

- unifiable(X,Y,Unifier)
  - If X and Y are unifiable, Unifier is a list of Var = Value pairs representing the bindings required to unify X and Y.

```
?- unifiable(f(X,Y),f(A,c), Unifier),
   writeln( (f(X,Y),f(A,c)) ).

f(_G11359, _G11360), f(_G11362, c)
Unifier = [Y=c, A=X].
```

  - This predicate can handle cyclic terms:

```
?- unifiable(f(X,Y),X, Unifier),
   writeln( (f(X,Y),X) ).

f(_G11299, _G11300), _G11299
Unifier = [X=f(X, Y)].
```

Testing Term Types

- var/1, nonvar/1
  - argument is a free variable
- integer/1, float/1, rational/1, number/1
  - rational includes integer, number is an integer or float
- atom/1
  - argument is an atom -- e.g. abc, 'John', …
- atomic/1
  - argument is an atom or number
- compound/1
  - argument is a compound term -- e.g. f(X,a)
- ground/1
  - argument contains no free variables -- e.g. 1, a, f(X,a) with X bound.
- cyclic_term/1, acyclic_term/1
  - argument does (not) contain cycles

Input and Output

File names and aliases

Prolog files

Explicit file I/O via streams (“ISO style”)

Implicit file I/O (“Edinburgh style”)

Implicit file I/O (“SWI style”)

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
- use_module(+File)
  - Compile and load prolog file that contains a “module”
  - Import all predicates from the export list of that module into the current module.
  - Note: The argument is the file name, not the module name!
- make/0
  - Check which loaded files have been modified and reconsult them

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.

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 ➢ 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	

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`.
- The current output stream is
  - obtained using `telling/1`
  - changed using `tell/1` or `append/1`.
- 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.
```

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(OldInput),tell(Output),   // set new output
     copy_implicit_io,                // do the real copying
    seen, see(OldInput),              // reset old input
    told, telling(OldOutput).         // reset old output

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

- Recall that the idiom used in `copy_implicit_io/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

Exception Handling

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

```
% Search a term in a file, succeed as soon as it 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
    -> !, fail
    ;  T = Term
    ).
```

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