Programming Paradigms
Unit 8 — Prolog Structures and Lists

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE
Outline

1. Structures
2. Equality, Matching and Arithmetic
3. Lists
4. Examples
Outline

1 Structures
2 Equality, Matching and Arithmetic
3 Lists
4 Examples
If we want to say that Wallace and Wendolene own books, we could formulate the following facts:

\[ \text{owns(wallace, book).} \]
\[ \text{owns(wendolene, book).} \]

However, this means that Wallace owns the same object that Wendolene owns.

Specifying the title to distinguish may not help:

\[ \text{owns(wallace, perfume).} \]
\[ \text{owns(wendolene, russell_the_sheep).} \]

It’s not clear that we are talking about books here.

This can be solved by introducing a \text{structure} for books.
A **structure** in Prolog is a single object which consists of a collection of other objects, called components.

A structure can be decomposed into:
- a **functor** and
- one or more **components**

The functor names the **general kind of structure**, and corresponds to a data type in other languages.

Using a structure for books, we have:

- `owns(wallace, book(perfume,suesskind)).`
- `owns(wendolene, book(russell_the_sheep,scotton)).`

Looking at `book(perfume,suesskind)`:
- `book` is the **functor** of the structure.
- `perfume` and `suesskind` are its **components**.
Nested Structures

- Structures can be **nested** (arbitrarily deep)
- Since there were three Brontë writers, we might want to present the author in more detail with another structure `author`, e.g.,

  `owns(gromit, book(wuthering_heights, author(emily,bronte))).`

- Prolog allows you to create arbitrarily complex structures to represent information/knowledge
- We could improve the book structure by adding an additional argument to represent *which copy* the book was
  - e.g., the third argument uniquely identifies the book

  `owns(gromit, book(wuthering_heights, author(emily,bronte), 3129)).`
Querying Structures

- Structures my participate in query processing, including the use of variables.
- For example, if we want to know if Gromit owns any books written by one of the Brontë sisters, we would query:

```prolog
?- owns(gromit, book(X, author(Y, bronte))).
X = wuthering_heights
Y = emily
```

That is: structures are matched similar as goals.

- from left to right
- functor names are literally matched (such as predicates)
- components: atoms are literally matched, variables match everything
- recursive matching of recursive structures
Structures and Facts

- The syntax for structures and facts is identical
  - A predicate (used in facts and rules) is actually the functor of a structure
  - The arguments of a fact or rule are components of a structure
- So, Prolog programs are essentially structures, which has several advantages.
- All parts of Prolog, even Prolog programs themselves, are made up of constants, variables and structures.
Outline

1. Structures
2. Equality, Matching and Arithmetic
3. Lists
4. Examples
Prolog has a number of built-in predicates
One of them is equality written as “=”
The expression \( X = Y \) attempts to match \( X \) and \( Y \)
  i.e., tries to make \( X \) and \( Y \) equal
The goal succeeds if \( X \) and \( Y \) match; otherwise it fails
Following Prolog syntax, it should be written as \( = (X, Y) \)
  While this works, Prolog also allows you to use an infix notation: \( X = Y \)
Equality and Matching of Atoms and Numbers

- Integers and atoms are always equal to themselves

?- wallace = wallace.
yes
?- cheese = cake.
no
?- 1066 = 1066.
yes
?- 1206 = 1583.
no
Equality and Matching of Variables/1

A variable always matches itself, i.e., \( X = X \) always succeeds

?- \( X = X \).
yes

If we match two different variables, i.e., \( X = Y \), we have to distinguish three cases

Case 1: none of the variables is instantiated
- The goal always succeeds

?- \( X = Y \).
\( X = Y \)
yes
Case 2: one of the two variables (say $Y$) is instantiated

- Goal succeeds, and $X$ is instantiated with the value of $Y$

?- $X = \text{gromit}$.
$X = \text{gromit}$.

?- $X = \text{likes(wallace, toast)}$.
$X = \text{likes(wallace, toast)}$.

?- $X = Y$, $\text{likes(X, toast)}$.
$X = \text{wallace}$.
$Y = \text{wallace}$.

Knowledge base

likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
Case 3: both variables are already instantiated
- The values the two variables are instantiated with are compared
- Might require the comparison of structures

?- likes(X,cheese), likes(Y,cake), X = Y.
X = gromit
Y = gromit

?- likes(X,toast), likes(Y,cake), X = Y.
no

Knowledge base
likes(wallace, toast).
likes(wallace, cheese).
likes(gromit, cheese).
likes(gromit, cake).
likes(wendolene, sheep).
Equality and Matching of Structures

- Two structures are equal if
  - they have the same functor and number of components and
  - all the corresponding components are equal

?- likes(gromit,cheese) = likes(gromit,X).
X = cheese

?- f(a,g(a,b)) = f(X,g(Y,Z)).
X = Y, Y = a,
Z = b.

?- a(b,C,d(e,F,g(h,i,J))) = a(B,c,d(E,f,g(H,i,j)))
B = b
C = c
E = e
F = f
H = h
J = j

?- letter(c) = word(c).
no.
Comparison and Matching

- Prolog also offers other built-in comparison operators
  
  ```prolog
  ?- 2 > 3.
  no
  ?- 3 >= 2.
  yes
  ?- 3 =< 2.
  no
  ?- X \= Y.
  no
  ```

- The \= operator means that X cannot be made equal to Y
  
  - `not(X=Y)` could also be used
Arithmetic

- Prolog also offers the standard arithmetic operators: +, -, *, /, mod,
- Just typing in an arithmetic operation will not actually carry it out
  ?- 3 + 4.
  ERROR: toplevel: Undefined procedure: (+)/2 ...
  ?- X = 3 + 4.
  X = 3 + 4.
  ?- 7 = 3 + 4.
  no

- Using the is operator will evaluate the right-hand side and match it to the left-hand side
  ?- 7 is 3 + 4.
  yes
  ?- X is 3 + 4.
  X = 7.
Arithmetic Example/1

Given the following fact base, compute the population density of countries

\[
\begin{align*}
\text{pop(usa, 313).} \\
\text{pop(italy, 61).} \\
\text{pop(uk, 63).} \\
\text{area(usa, 9.826).} \\
\text{area(italy, 0.301).} \\
\text{area(uk, 0.243).}
\end{align*}
\]

The following rule computes the density

\[
\text{density(X, Y) :- pop(X, P), area(X, A), } \quad Y \text{ is } P/A.
\]

This rule is read as follows:

- The population density of country X is Y, if:
  - The population of X is P, and
  - The area of X is A, and
  - Y is calculated by dividing P by A.
Arithmetic Example/2

Compute population density of USA

?- density(usa,Y).
Y = 31.854264197028289
yes

Compute all densities

?- density(X,Y).
X = usa
Y = 31.854264197028289 ;
X = italy
Y = 202.65780730897012 ;
X = uk
Y = 259.25925925925924.
Outline

1. Structures
2. Equality, Matching and Arithmetic
3. Lists
4. Examples
We have already seen structures as a construct to build more complicated data types.

Another important type supported by Prolog is a **list**

The elements of a list are enclosed in square brackets `[ ]`

?- [1,2,3] = [1,2,3].
yes

?- [1,2,3] = [X,Y,Z].
X = 1,
Y = 2,
Z = 3.

Lists are matched similar to structures
The elements of a list can be *any* terms – constants, variables, structures, lists

... and they can be *mixed*

Examples of valid lists are

- `[]`
- `[2,3,5,a,b]`
- `[the,cat,sat,[on,the,mat]]`
- `[a,V,b,[X,Y]]`
- `[the,book([programming,in,prolog]),by,authors(C,M)]`
**Internal Representation of Lists**

- Internally, **lists are represented as compound terms using**
  - the functor "."/2 (dot, list constructor), where the first argument is the first element and the second argument is the rest of the list, and
  - the **atom []** representing the empty list, which is the second argument on the innermost level.

- For example, the list
  
  \[a,b,c]\n
  corresponds to the compound term
  
  \(.\underline{a}, \underline{(b, \underline{(c, []))})\)

- So, \[1,2,3\] is just a more convenient notation for an important structure

- We can verify this in Prolog:
  
  ?- X = .(a, .(b, .(c, []))).
  X = [a, b, c]
  Yes

- In SWI Prolog v7, the functor "." has been replaced by the functor "[]"
We can split lists into a head and tail using the "|" operator

- Head is the first element of the list
- Tail is the (possibly empty) rest of the list, and it is a list

?- [Head|Tail] = [1,2,3].
Head = 1,
Tail = [2,3].

?- [Head|Tail] = [].
no.

?- [Head|Tail] = [1].
Head = 1,
Tail = [].

Splitting Lists in Head and Tail/1
Some more examples

?- [H|T] = [[the,cat],sat].
H = [the,cat],
T = [sat].

?- [H|T] = [the,[[cat,sat],down]].
H = the,
T = [[cat,sat],down].

?- [H|T] = [X+Y,x+y].
H = X+Y,
T = [x+y].
Lists and Recursion/1

- Let's assume we want to find out if an element is part of a list
  - Prolog has the built-in predicate `member(X,Y)`, but define our own predicate
- We have to do this recursively in Prolog
  - There are no loops like in other programming languages
- Recursion in Prolog means that a predicate appears on the left- and the right-hand side of a rule
- For example, *an element is in a list* if it is
  - *the head* of the list or
  - *in the tail* of the list

\[
is\_in(X,[H|\_]) :- X = H.
\]
\[
is\_in(X,[\_|T]) :- is\_in(X,T).
\]

?- is\_in(d,[a,b,c,d,e,f]).
true
Step-by-step execution of the goal on the previous slide

\[
\text{is\_in}(X,[H|\_]) :\!- X = H. \\
\text{is\_in}(X,[_|T]) :\!- \text{is\_in}(X,T).
\]

?\- \text{is\_in}(d,[a,b,c,d,e,f]).
true

<table>
<thead>
<tr>
<th>(Recursive) call</th>
<th>Rule 1</th>
<th>Rule 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{is_in}(d,[a,b,c,d,e,f])</td>
<td>(X = d, H = a \rightarrow \text{false})</td>
<td>(X = d, T = [b,c,d,e,f])</td>
</tr>
<tr>
<td>\text{is_in}(d,[b,c,d,e,f])</td>
<td>(X = d, H = b \rightarrow \text{false})</td>
<td>(X = d, T = [c,d,e,f])</td>
</tr>
<tr>
<td>\text{is_in}(d,[c,d,e,f])</td>
<td>(X = d, H = c \rightarrow \text{false})</td>
<td>(X = d, T = [d,e,f])</td>
</tr>
<tr>
<td>\text{is_in}(d,[d,e,f])</td>
<td>(X = d, H = d \rightarrow \text{true})</td>
<td></td>
</tr>
</tbody>
</table>
Lists and Recursion/3

- Does the is_in predicate cover all cases?
- Having a closer look at the recursion, we observe that there are actually two base cases for the is_in predicate
  - Base case 1: element X is the head of the list (first predicate)
  - Base case 2: element X is not in the list, then the list is empty
- However, the second base case need not to be implemented, as none of the two predicates matches an empty list as second parameter
- However, we could add the following clause for the second base case

  \[\text{is\_in(X,L) :- L = [], fail.}\]

  - Predicate \text{fail} returns false
- The termination of is_in is guaranteed as in the recursive call the list passed to the goal is shorter, hence
  - eventually X is encountered as first element of the list (base case 1),
  - or the list is empty (base case 2)
Enumerating Elements and Generating Lists

- The predicate `is_in` can also be used to enumerate all elements of a list

```prolog
?- is_in(X,[1,2,a]).
X = 1;
X = 2;
X = a;
false
```

- We can even use it to generate lists

```prolog
?- is_in(a,L).
L = [a|_G5033893];
L = [_G5033893, a|_G5033898]
```

- `_G5033893, ...` are variables
List Predicates – last/2

Finding the last element of a list

\[
\text{last}(X,[X]).
\]
\[
\text{last}(X,[\_|T]) :- \text{last}(X,T).
\]

?- last(X,[\text{talk,of,the,town}]).

\[
X = \text{town}
\]
List Predicates – next_to/2

- Checking for two consecutive elements of a list

next_to(X,Y,[X,Y|_]).
next_to(X,Y,[_|Z]) :- next_to(X,Y,Z).

?- next_to(X,Y,[talk,of,the,town]).
X = talk,
Y = of ;
X = of ,
Y = the ;
X = the ,
Y = town
List Predicates – append/3

- **append** is a very useful built-in predicate that can be used in a flexible way
- **Appending** two lists

  ```prolog
  append([], L, L).
  append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
  ```

  ```prolog
  ?- append([i, like], [prolog], L).
  L = [i, like, prolog]
  ```

- **Generating** sublists

  ```prolog
  ?- append(X, Y, [i, like, prolog]).
  X = [], Y = [i, like, prolog] ;
  X = [i], Y = [like, prolog] ;
  X = [i, like], Y = [prolog] ;
  X = [i, like, prolog], Y = [] ;
  no
  ```

- **Computing the difference** between lists

  ```prolog
  ?- append([i], Y, [i, like, prolog]).
  Y = [like, prolog]
  ```
List Predicates – append/3

- Prolog is very flexible with regard to the initialization of parameters
  - e.g., in append any of two parameters can be initialized
- We can easily implement last, next_to, and is_in using append

last(E,List) :- append(_, [E], List).

next_to(X,Y,List) :- append(_, [X,Y|_], List).

is_in(X,List) :- append(_, [X|_], List).
Strings in Prolog can be quite confusing if you come from another language.

There are two “types” of strings:

Strings enclosed in single quotes ('') are atoms.

?- 'hello' = S.
S = hello.

write('hello')
hello
true.

As this class of strings are atoms, they naturally cannot be manipulated.
Strings in Prolog/2

- Strings (or terms) written in **double quotes (" ")** are immediately converted to a list of character codes (ASCII)

```prolog
?- "hello" = L.
write("hello").
true.
```

- As of SWI-Prolog v7, only **back quoted** text is converted,
  - e.g., ‘text’ is represented as [116,101,120,116],
  whereas text enclosed in double quotes is read as a sequence of characters
Strings in Prolog/3

- Sometimes, single-quoted strings need to be converted to character lists, e.g., to print the first character of a string or to search for a substring.
- This can be done by the `name` predicate.

  ```prolog
  ?- name('hello', L).
  ```

- SWI-Prolog provides a large number of built-in predicates for strings, e.g., concatenate strings, string length, conversion between terms and strings, etc.
Prefix Example

The following predicate verifies, whether a string \( S_1 \) is a prefix of another string \( S_2 \).

\[
\text{prefix}(S_1, S_2) :-
\text{atom}(S_1),
\text{atom}(S_2),
\text{name}(S_1, L_1),
\text{name}(S_2, L_2),
\text{append}(L_1, \_ , L_2).
\]

We can use it as follows:

\[
?- \text{prefix}('hello', 'hello world').
true.
\]

\[
?- \text{prefix}("hello", "hello world").
false.
\]
Outline

1 Structures

2 Equality, Matching and Arithmetic

3 Lists

4 Examples
The Towers of Hanoi

Goal: move a stack of \( n \) disks from one peg to another with the help of an auxiliary peg, where

- Only one disk can be moved at a time
- A move can only take the upper disk of a stack
- A larger disk can never be placed on top of a smaller disk

Legend: Somewhere in the surrounding of Hanoi, there is a monastery, where the monks have to perform this task assigned to them by God when the world was created with \( n = 64 \) golden disks. At the moment they complete their task, the world will collapse.

The minimum number of moves to solve a Tower of Hanoi puzzle is \( 2^n - 1 \)

This is roughly 585 billion years for the monks, if they move the disks at a rate of one move per second.
The Towers of Hanoi/2

- Recursive solution
  - Termination: there are no disks on peg A
  - Move \( n - 1 \) disks from peg A to C (notice the recursive move!)
  - Move disk \( n \) from peg A to B
  - Move \( n - 1 \) disks from peg C to B

- Predicate \( \text{Move}(N,A,B,C) \) moves \( n \) disks from peg A to peg B with the help of C

\[
\text{hanoi}(N) :- \text{move}(N,\text{pegA},\text{pegB},\text{pegC}).
\]

\[
\text{move}(1,A,B,\_):- \text{write}([\text{move, disc, from, A, to, B}]), \text{nl}.
\]
\[
\text{move}(N,A,B,C):-
\]
\[
N > 1,
M \equiv N-1,
\text{move}(M,A,C,B),
\text{move}(1,A,B,\_),
\text{move}(M,C,B,A).
\]
Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9x9 grid with digits so that each column, each row, and each of the nine 3x3 sub-grids (boxes) contains all of the digits from 1 to 9.

A partially completed grid is given, which has a unique solution.

![Sudoku Grid](image)
We make the problem easier and consider a 4x4 sudoku, where rows, columns and boxes have to be filled with a permutation of the numbers 1, ..., 4.

We can model the Sudoku problem in Prolog using list permutations:

- Each row must be a permutation of [1, 2, 3, 4]
- Each column must be a permutation of [1, 2, 3, 4]
- Each 2x2 box must be a permutation of [1, 2, 3, 4]

The Sudoku is represented by a list of lists:

```prolog
[[X11, X12, X13, X14],
 [X21, X22, X23, X24],
 [X31, X32, X33, X34],
 [X41, X42, X43, X44]]
```
sudoku([R1, R2, R3, R4]) :-
    R1 = [X11, X12, X13, X14],
    R2 = [X21, X22, X23, X24],
    R3 = [X31, X32, X33, X34],
    R4 = [X41, X42, X43, X44],
    % rows
    permutation([X11, X12, X13, X14], [1, 2, 3, 4]),
    permutation([X21, X22, X23, X24], [1, 2, 3, 4]),
    permutation([X31, X32, X33, X34], [1, 2, 3, 4]),
    permutation([X41, X42, X43, X44], [1, 2, 3, 4]),
    % cols
    permutation([X11, X21, X31, X41], [1, 2, 3, 4]),
    permutation([X12, X22, X32, X42], [1, 2, 3, 4]),
    permutation([X13, X23, X33, X43], [1, 2, 3, 4]),
    permutation([X14, X24, X34, X44], [1, 2, 3, 4]),
    % boxes
    permutation([X11, X12, X21, X22], [1, 2, 3, 4]),
    permutation([X13, X14, X23, X24], [1, 2, 3, 4]),
    permutation([X31, X32, X41, X42], [1, 2, 3, 4]),
    permutation([X33, X34, X43, X44], [1, 2, 3, 4]).
Binary Search Trees/1

- **Binary search trees** can be represented in Prolog by a recursive structure with three arguments \( \text{bst}(K,L,R) \), where
  - \( K \) is the key of the root
  - \( L \) is the left sub-tree
  - \( R \) is the right sub-tree

- The empty (null) tree is usually represented as the constant nil.

- Example tree with 6 nodes:
  
  \[
  \text{bst}(6,\
  \quad \text{bst}(4,\
  \quad \quad \text{bst}(2,\text{nil},\text{nil}),\
  \quad \quad \text{bst}(5,\text{nil},\text{nil})),\
  \quad \text{bst}(9,\
  \quad \quad \text{bst}(7, \text{nil}, \text{nil}),\
  \quad \quad \text{nil})\
  \)
  \]
Binary Search Trees/2

A basic operation is \texttt{bstmem(Tree,X)}, which succeeds if X is contained in Tree

\begin{verbatim}
bstmem(bst(X,_,_), X).
bstmem(bst(K,L,_), X) :-
    X < K,
    bstmem(L, X).
bstmem(bst(K,_,R), X) :-
    X > K,
    bstmem(R, X)
\end{verbatim}

Examples:

?- bstmem(nil, 3).
No
?- bstmem(bst(5,bst(8,nil,nil),nil),8).
Yes
Another basic operation is \texttt{inorder(Tree, L)} that succeeds if \(L\) contains the keys in \(Tree\) in inorder.

\[
\text{inorder(bst(K,L,R), List) :- }
\text{inorder(L, LL),}
\text{inorder(R, LR),}
\text{append(LL, [K|LR], List).}
\text{inorder(nil, []).}
\]

Examples:

\?	ext{- inorder(bst(5,bst(4,nil,nil),bst(8,nil,nil)),L).}
L = [4,5,8]

Modify the above predicate to a predicate

\bullet \text{ preorder(Tree,L) that succeeds if }L\text{ contains the keys in Tree in preorder}
\bullet \text{ postorder(Tree,L) that succeeds if }L\text{ contains the keys in Tree in postorder}