DTU Course 02156 Logical Systems and Logic Programming (2021)

Week	Date	Main Topics (Prolog Programming in All Lessons)
35 #01	31/8	Course Prerequisites & Tutorial on Logical Systems and Logic Programming
36 #02	7/9	Chapter 1 - Introduction (Prolog Note)
37 #03	14/9	Chapter 2 - Propositional Logic: Formulas, Models, Tableaux
38 #04	21/9	Chapter 3 - Propositional Logic: Deductive Systems
39 #05	28/9	"Isabelle" - Propositional Logic: Sequent Calculus Verifier (SeCaV)
40 #06	5/10	Chapter 4 - Propositional Logic: Resolution
41 #07	12/10	Chapter 7 - First-Order Logic: Formulas, Models, Tableaux
42		(Autumn Vacation)
43 #08	26/10	Chapter 8 - First-Order Logic: Deductive Systems
44 #09	2/11	"Isabelle" - First-Order Logic: Sequent Calculus Verifier (SeCaV)
45 #10	9/11	Chapter 9 - First-Order Logic: Terms and Normal Forms
46 #11	16/11	Chapter 10 - First-Order Logic: Resolution
47 #12	23/11	Chapter 11 - First-Order Logic: Logic Programming
48 #13	30/11	Chapter 12 - First-Order Logic: Undecidability and Model Theory & Course Evaluation

Responsible: Associate Professor Jørgen Villadsen <jovi@dtu.dk>

Assignments & Exam **MUST BE SOLVED INDIVIDUALLY**

Assignment-1 Deadline Sunday 26/9 (Available Wednesday 15/9)

Assignment-2 Deadline Sunday 10/10 (Available Wednesday 29/9)

Assignment-3 Deadline Sunday 31/10 (Available Wednesday 13/10)

Assignment-4 Deadline Sunday 14/11 (Available Wednesday 3/11)

Assignment-5 Deadline Thursday 2/12 (Available Wednesday 17/11)

Written Exam Tuesday 14/12 (2 Hours / No Computer / All Notes Allowed)

The mandatory assignments and the written exam are evaluated as a whole – even if you do well in the mandatory assignments then you still must do decent in the written exam in order to pass the course!

A TEACHER MUST IMMEDIATELY REPORT ANY SUSPICION OF CHEATING TO THE STUDY ADMINISTRATION FOR FURTHER ACTIONS

Propositional Logic: Truth Tables Clavius's Law $(\neg A \rightarrow A) \rightarrow A$

Tautology

If there are no true sentences then there are true sentence

(for instant the sentence: there are no true sentences)

So there are true sentences

Ludwig Wittgenstein

1889 – 1951

https://en.wikipedia.org/wiki/Ludwig_Wittgenstein

Christopher Clavius

1538 – 1612

https://en.wikipedia.org/wiki/Christopher_Clavius

Clavius's Law = Consequentia Mirabilis (Latin: "admirable consequence")

https://en.wikipedia.org/wiki/Consequentia_mirabilis

Agenda — Week #2

About the course

Prolog note (pages 1-8)

Tracer — More Next Week

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Content

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Content

Logic programming, in particular Prolog as a rapid prototyping tool

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Content

Logic programming, in particular Prolog as a rapid prototyping tool

Elementary logics, including propositional and first-order logics

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Content

Logic programming, in particular Prolog as a rapid prototyping tool

Elementary logics, including propositional and first-order logics

Proof systems, deductive systems and/or refutation systems

General Course Objectives

The aim of the course is to give the students an introduction to some of the basic declarative formalisms from formal computer science that can be used for describing, analysing and evaluating aspects of IT-systems.

It will cover theoretical insight as well as practical skills in relevant high-level programming languages.

Content

Logic programming, in particular Prolog as a rapid prototyping tool

Elementary logics, including propositional and first-order logics

Proof systems, deductive systems and/or refutation systems

Problem solving techniques, like the backtracking algorithm

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Tryg-Baltica A/S (1994-1996) *

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Tryg-Baltica A/S (1994-1996) *

Centre for Language Technology (1992-1994)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Tryg-Baltica A/S (1994-1996) *

Centre for Language Technology (1992-1994)

PhD & MSc, DTU Computer Science (1984-1992)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Tryg-Baltica A/S (1994-1996) *

Centre for Language Technology (1992-1994)

PhD & MSc, DTU Computer Science (1984-1992)

Lots of Prolog programming in academia and industry :-)

Jørgen Villadsen

Associate Professor, DTU (2006-...)

Associate Professor, Roskilde University (2002-2006)

Assistant Professor, DTU (1999-2002)

Prolog Development Center A/S (1999)

Danish Defence Research Establishment (1997-1999)

Tryg-Baltica A/S (1994-1996) *

Centre for Language Technology (1992-1994)

PhD & MSc, DTU Computer Science (1984-1992)

Lots of Prolog programming in academia and industry :-)

— except *

ISO Prolog

The note is a summary of the logic programming language Prolog.

ISO Prolog

The note is a summary of the logic programming language Prolog.

There are many different Prolog systems today.
The ISO standard for Prolog defines a large set of predicates.
The predicates described in the following belong to ISO Prolog unless otherwise noted.

ISO Prolog

The note is a summary of the logic programming language Prolog.

There are many different Prolog systems today.
The ISO standard for Prolog defines a large set of predicates.
The predicates described in the following belong to ISO Prolog unless otherwise noted.

A Prolog system like SWI-Prolog displays a prompt `?-` and waits for a query, for example (the dot `.` marks the end of the query):

```
?- halt.
```

The predicate `halt` terminates the Prolog system immediately and is therefore rarely used.

ISO Prolog

The note is a summary of the logic programming language Prolog.

There are many different Prolog systems today.
The ISO standard for Prolog defines a large set of predicates.
The predicates described in the following belong to ISO Prolog unless otherwise noted.

A Prolog system like SWI-Prolog displays a prompt `?-` and waits for a query, for example (the dot `.` marks the end of the query):

```
?- halt.
```

The predicate `halt` terminates the Prolog system immediately and is therefore rarely used.

Except for special queries with a predicate like `halt` and queries that loop forever it is characteristic of queries that they either succeed or fail.

Succeed

Here is a query that succeeds:

```
?- write('Hello World'), nl.
Hello World


Yes
```

Recent versions of the SWI-Prolog system uses `true` instead of `Yes` and `false` instead of `No` (a dot `.` is also added at the end).

Succeed

Here is a query that succeeds:

```
?- write('Hello World'), nl.
Hello World


Yes
```

Recent versions of the SWI-Prolog system uses `true` instead of `Yes` and `false` instead of `No` (a dot `.` is also added at the end).

The program writes the greeting and a new line.
The word `Yes` means that the query succeeds.

Succeed

Here is a query that succeeds:

```
?- write('Hello World'), nl.
Hello World


Yes
```

Recent versions of the SWI-Prolog system uses `true` instead of `Yes` and `false` instead of `No` (a dot `.` is also added at the end).

The program writes the greeting and a new line.
The word `Yes` means that the query succeeds.

Whether queries succeed or fail have nothing to do with errors or exceptions.
The `,` (comma) means "and" (logical conjunction) and `;` (semicolon) means "or" (logical disjunction to be used later).

Fail

Here is a query that fails:

```
?- write('Hello World'), nl, fail.
Hello World

No
```

Fail

Here is a query that fails:

```
?- write('Hello World'), nl, fail.
Hello World

No
```

The word `No` means that the query fails.
Observe that the greeting is written before the failure.

Fail

Here is a query that fails:

```
?- write('Hello World'), nl, fail.
Hello World

No
```

The word `No` means that the query fails.
Observe that the greeting is written before the failure.

The following query fails before the greeting is written:

```
?- fail, write('Hello World'), nl.

No
```

The predicate `fail` always fails, but is nevertheless quite useful :-)

Succeed More Than Once

Predicates like `write` and `nl` succeed only once and they are normally used only in special program blocks, if at all.

Succeed More Than Once

Predicates like `write` and `nl` succeed only once and they are normally used only in special program blocks, if at all.

Many other predicates succeed more than once and the extreme is the predicate `repeat` that succeeds an unlimited number of times. The following query will loop forever:

```
?- repeat, fail.
```

The predicate `true` succeeds once.

Succeed More Than Once

Predicates like `write` and `nl` succeed only once and they are normally used only in special program blocks, if at all.

Many other predicates succeed more than once and the extreme is the predicate `repeat` that succeeds an unlimited number of times. The following query will loop forever:

```
?- repeat, fail.
```

The predicate `true` succeeds once.

The use of the predicates `true` and `repeat` is not recommended for novices.

Variables

A variable should always start with an uppercase letter (`A` ... `Z`) and the remaining letters in variables can be lowercase letters and/or numbers.

```
X   H   T   Args   ArgsRest   V123
```

An exception is the special anonymous variable `_` (underscore) that simply is a new variable for each occurrence.

Variables

A variable should always start with an uppercase letter (`A` ... `Z`) and the remaining letters in variables can be lowercase letters and/or numbers.

```
X   H   T   Args   ArgsRest   V123
```

An exception is the special anonymous variable `_` (underscore) that simply is a new variable for each occurrence.

There is no type system in Prolog, hence a variable can hold any kind of data.

Constants

Although there is no type system in Prolog there are different kinds of constants, in particular atoms like `a` and numbers like 2 (decimal numbers are also possible).

Constants

Although there is no type system in Prolog there are different kinds of constants, in particular atoms like `a` and numbers like `2` (decimal numbers are also possible).

An atom that starts with an uppercase letter, or contains special characters like spaces, must be in quotes, like `'Hello World'` in the examples.

Constants

Although there is no type system in Prolog there are different kinds of constants, in particular atoms like `a` and numbers like `2` (decimal numbers are also possible).

An atom that starts with an uppercase letter, or contains special characters like spaces, must be in quotes, like `'Hello World'` in the examples.

Constants are terms (so are variables), and compound terms can easily be formed:

```
f(1,2,3)   tree(tree(nil,nil),nil)   (1,2)   [a,b,c]
```

Here `f` and `tree` are called functors (atoms are also functors).

Constants

Although there is no type system in Prolog there are different kinds of constants, in particular atoms like `a` and numbers like 2 (decimal numbers are also possible).

An atom that starts with an uppercase letter, or contains special characters like spaces, must be in quotes, like `'Hello World'` in the examples.

Constants are terms (so are variables), and compound terms can easily be formed:

```
f(1,2,3)   tree(tree(nil,nil),nil)   (1,2)   [a,b,c]
```

Here `f` and `tree` are called functors (atoms are also functors).

Again no types are given or inferred, and there are no restrictions on elements of a list:

```
[1,a,g([]),nil]
```

Resolution

Prolog uses the so-called SLD-resolution procedure.

Resolution

Prolog uses the so-called SLD-resolution procedure.

The acronym SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses.

Resolution

Prolog uses the so-called SLD-resolution procedure.

The acronym SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses.

A logic program is a sequence of definite clauses.

Resolution

Prolog uses the so-called SLD-resolution procedure.

The acronym SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses.

A logic program is a sequence of definite clauses.

Such definite clauses are also called facts and program clauses and a query is called a goal clause.

Resolution

Prolog uses the so-called SLD-resolution procedure.

The acronym SLD stands for Selecting a literal, using a Linear strategy, restricted to Definite clauses.

A logic program is a sequence of definite clauses.

Such definite clauses are also called facts and program clauses and a query is called a goal clause.

The answer to a query is a set of substitutions:

```
?- length([a,b,c],N).

N = 3

Yes
```

The answer means that the query succeeds with 3 substituted for `N`.

Equality

The infix predicate = is simply defined as follows:

```
X = X.
```

It is completely symmetric and arithmetic expressions are not evaluated (but the term 2+2 is equal to itself).

```
?- 2+2 = 2+2, X = 2+2, 2+2 = Y.

X = 2+2
Y = 2+2 ;

No
```

In SWI-Prolog a `;` (semicolon) can be entered after the answer in order to search for other answers, but there are no other way that the query can succeed in this case as indicated by the `No` for failure.

Unification

Prolog uses unification which means that variables can appear anywhere in terms:

```
?- tree(tree(nil,A),B) = tree(C,tree(nil,nil)).

B = tree(nil, nil)
C = tree(nil, A)

Yes
```

Here no ; (semicolon) was entered after the first and only answer.

Unification

Prolog uses unification which means that variables can appear anywhere in terms:

```
?- tree(tree(nil,A),B) = tree(C,tree(nil,nil)).

B = tree(nil, nil)
C = tree(nil, A)

Yes
```

Here no ; (semicolon) was entered after the first and only answer.

Note that nothing is substituted for `A`.

Unification

Prolog uses unification which means that variables can appear anywhere in terms:

```
?- tree(tree(nil,A),B) = tree(C,tree(nil,nil)).

B = tree(nil, nil)
C = tree(nil, A)

Yes
```

Here no ; (semicolon) was entered after the first and only answer.

Note that nothing is substituted for `A`.

This means that anything can be substituted for `A`.

Unification

Prolog uses unification which means that variables can appear anywhere in terms:

```
?- tree(tree(nil,A),B) = tree(C,tree(nil,nil)).

B = tree(nil, nil)
C = tree(nil, A)

Yes
```

Here no ; (semicolon) was entered after the first and only answer.

Note that nothing is substituted for `A`.

This means that anything can be substituted for `A`.

Variables like `A` are called logical variables.

Lists

Special notation for lists with head and tail:

```
?- L = [X,Y,Z], [H|T] = L, L = [a,b,c], [A,B|R] = L.

L = [a, b, c]
X = a
Y = b
Z = c
H = a
T = [b, c]
A = a
B = b
R = [c] ;

No
```

Note that `A` and `B` are the first two elements in the list.

Arithmetic

Arithmetic expressions allow the usual operators `+`, `-`, `*`, and `/` (division may return a decimal number for integer arguments).

Arithmetic

Arithmetic expressions allow the usual operators `+`, `-`, `*`, and `/` (division may return a decimal number for integer arguments).

The following special infix predicates evaluate `A` and `B`

`A < B`	`A > B`	Less than / Greater than
`A =< B`	`A >= B`	Equal or less than / Greater than or equal
`A =:= B`	`A =\= B`	Equal / Not equal
`X is A`		Unification with `X`

For example (here `//` is integer division):

```
?- X is (2+3)//2.

X = 2 ;

No
```

Program

A program is a sequence of clauses: *Head* `:-` *Body*`.`

If *Body* is empty then `:-` is omitted.

Program

A program is a sequence of clauses: *Head* `:-` *Body*`.`
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

Program

A program is a sequence of clauses: *Head* `:-` *Body*`.`
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

Program

A program is a sequence of clauses: *Head* `:-` *Body*`.`
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.`   `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

The functor of *Head* is the predicate being defined (here `p`).
The *Body* is a query (here `q, r ; s, t`) or can be empty.

Program

A program is a sequence of clauses: *Head* `:-` *Body*`.`
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

The functor of *Head* is the predicate being defined (here `p`).
The *Body* is a query (here `q, r ; s, t`) or can be empty.

The operator `,` (comma) means "and" in propositional logic ($\wedge$).

Program

A program is a sequence of clauses: *Head* `:-` *Body*.
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

The functor of *Head* is the predicate being defined (here `p`).
The *Body* is a query (here `q, r ; s, t`) or can be empty.

The operator `,` (comma) means "and" in propositional logic ($\wedge$).

The operator `;` (semicolon) means "or" in propositional logic ($\vee$).

Program

A program is a sequence of clauses: *Head* `:-` *Body*.
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

The functor of *Head* is the predicate being defined (here `p`).
The *Body* is a query (here `q, r ; s, t`) or can be empty.

The operator `,` (comma) means "and" in propositional logic ($\wedge$).

The operator `;` (semicolon) means "or" in propositional logic ($\vee$).

The "and" binds tighter than "or" as usual.

Program

A program is a sequence of clauses: *Head* `:-` *Body*.
If *Body* is empty then `:-` is omitted.

Here are two clauses: `p.` `p :- q, r ; s, t.`

The operator `:-` means "if" in propositional logic (symbol $\leftarrow$ which is simply $\rightarrow$ with the arguments exchanged).

The functor of *Head* is the predicate being defined (here `p`).
The *Body* is a query (here `q, r ; s, t`) or can be empty.

The operator `,` (comma) means "and" in propositional logic ($\wedge$).

The operator `;` (semicolon) means "or" in propositional logic ($\vee$).

The "and" binds tighter than "or" as usual.

Hence the second clause is really: `p :- (q, r) ; (s, t).`
The use of "or" is not recommended for novices and can always be avoided by additional clauses and/or predicates.

A Sample Program — To Be Revised Later

Here is a program that writes the elements of a list in reverse order:

```
main(Args) :-
  Args = []
  ;
  Args = [Arg|ArgsRest],
  main(ArgsRest), write(Arg), nl.
```

A Sample Program — To Be Revised Later

Here is a program that writes the elements of a list in reverse order:

```
main(Args) :-
  Args = []
  ;
  Args = [Arg|ArgsRest],
  main(ArgsRest), write(Arg), nl.
```

Predicate `main` has no reserved role in ISO Prolog.

```
?- main([a,b,c]).
c
b
a

Yes
```

Conventions

Comments can be a block `/* ...*/` or the rest of the line `% ...`

Conventions

Comments can be a block `/* ... */` or the rest of the line `% ...`

They are often used to specify instantiation patterns for predicates:

```
main(+List)
```

Arguments are preceded by a sign where "+" indicates that the argument is input to the predicate, "–" indicates output, and "?" indicates either input or output.

Conventions

Comments can be a block `/* ... */` or the rest of the line `% ...`

They are often used to specify instantiation patterns for predicates:

```
main(+List)
```

Arguments are preceded by a sign where "+" indicates that the argument is input to the predicate, "-" indicates output, and "?" indicates either input or output.

The number of arguments is called the arity of the predicate. Often the arity is given together with the predicate: `main/1`

Conventions

Comments can be a block `/* ... */` or the rest of the line `% ...`

They are often used to specify instantiation patterns for predicates:

```
main(+List)
```

Arguments are preceded by a sign where "+" indicates that the argument is input to the predicate, "–" indicates output, and "?" indicates either input or output.

The number of arguments is called the arity of the predicate. Often the arity is given together with the predicate: `main/1`

This makes good sense since due to the lack of a type system the instantiation patterns are just conventions, whereas predicates with different arities are always distinguished.

Conventions

Comments can be a block `/* ... */` or the rest of the line `% ...`

They are often used to specify instantiation patterns for predicates:

```
main(+List)
```

Arguments are preceded by a sign where "+" indicates that the argument is input to the predicate, "-" indicates output, and "?" indicates either input or output.

The number of arguments is called the arity of the predicate. Often the arity is given together with the predicate: `main/1`

This makes good sense since due to the lack of a type system the instantiation patterns are just conventions, whereas predicates with different arities are always distinguished.

The arity can also be used for functors: `tree/2`, `nil/0`, etc.

A Sample Program — Revised

Same program in the recommended style:

```
% Prolog file main.pl

/*
main(+List)

Writes the elements of List in reverse order.
*/

main([]).
main([Arg|ArgsRest]) :-
  main(ArgsRest),
  write(Arg),
  nl.
```

Length

Consider a predicate `length` that can be used to calculate the number of elements in a list.

```
length(+List,?Integer)
```

For example:

```
?- length([a,b,c],N).

N = 3 ;

No
```

SWI-Prolog has a flexible built-in `length` predicate:

```
length(?List,?Integer)
```

Length Problems

The following program works for the simple `length(+List,?Integer)` instantiation pattern:

```
length([],0).
length([_|T],N1) :- length(T,N), N1 is N+1.
```

However it loops forever for the flexible `length(?List,?Integer)` instantiation pattern:

```
?- length(X,-1).
```

The built-in `length` predicate in SWI-Prolog uses advanced features to avoid these problems.

Length Problems

The following program works for the simple `length(+List,?Integer)` instantiation pattern:

```
length([],0).
length([_|T],N1) :- length(T,N), N1 is N+1.
```

However it loops forever for the flexible `length(?List,?Integer)` instantiation pattern:

```
?- length(X,-1).
```

The built-in `length` predicate in SWI-Prolog uses advanced features to avoid these problems.

The `length` predicate is not in ISO Prolog.

Auto-Loaded Predicates

In SWI-Prolog the predicate `length(?List,?Integer)` is built-in, which means that it cannot be redefined.

In SWI-Prolog the basic predicates `member(?Elem,?List)` and `append(?List,?List,?List)` are library auto-loaded, which means that they can be redefined, but are otherwise like built-in predicates.

None of `length`, `member` and `append` are ISO Prolog predicates.

By the way, `sort(+List,?Sorted)` is a built-in ISO predicate for sorting a list (duplicates are removed):

```
?- sort([3,1,4,1,2],S).

S = [1, 2, 3, 4]

Yes
```

Member

```
/*

member(?Elem,?List)

Succeeds iff Elem can be unified with
one of the members of List.

*/

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
```

Member

```
/*

member(?Elem,?List)

Succeeds iff Elem can be unified with
one of the members of List.

*/

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
```

For example `member(b,[a,b,c])` succeeds.

Member

```
/*

member(?Elem,?List)

Succeeds iff Elem can be unified with
one of the members of List.

*/

member(H,[H|_]).
member(H,[_|T]) :- member(H,T).
```

For example `member(b,[a,b,c])` succeeds.

The `member` predicate is not in ISO Prolog.

Append

```
/*

append(?List1,?List2,?List3)

Succeeds iff List3 unifies with
the concatenation of List1 and List2.

*/

append([],U,U).
append([H|T],U,[H|V]) :- append(T,U,V).
```

Append

```
/*

append(?List1,?List2,?List3)

Succeeds iff List3 unifies with
the concatenation of List1 and List2.

*/

append([],U,U).
append([H|T],U,[H|V]) :- append(T,U,V).
```

For example `append([a,b],[c,d],[a,b,c,d])` succeeds.

Append

```
/*

append(?List1,?List2,?List3)

Succeeds iff List3 unifies with
the concatenation of List1 and List2.

*/

append([],U,U).
append([H|T],U,[H|V]) :- append(T,U,V).
```

For example `append([a,b],[c,d],[a,b,c,d])` succeeds.

The `append` predicate is not in ISO Prolog.

Examples 1

```
?- member(b,[a,b,c]).

Yes

?- member(a,[b,c,d]).

No

?- member(X,[a,b,c]), member(X,[b,c,d]).

X = b ;

X = c ;

No
```

Examples 2

```
?- append(X,Y,[a,b,c]).

X = []
Y = [a, b, c] ;

X = [a]
Y = [b, c] ;

X = [a, b]
Y = [c] ;

X = [a, b, c]
Y = [] ;

No
```

Examples 3

```
?- append([a,b],Y,[a,_,c]).

Y = [c]

Yes

?- append(X,Y,Z).

X = []
Y = Z

Yes
```

The predicates `member/2` and `append/3` can be used with any instantiation patterns.

Tracer — More Next Week

```
?- trace, member(X,[a,b]).
```

Tracer — More Next Week

```
?- trace, member(X,[a,b]).
   Call: (1) member(_0,[a,b]) ?
   Exit: (1) member(a,[a,b]) ?

X = a ;

   Redo: (1) member(_0,[a,b]) ?
   Call: (2) member(_0,[b]) ?
   Exit: (2) member(b,[b]) ?
   Exit: (1) member(b,[a,b]) ?

X = b ;

   Redo: (2) member(_0,[b]) ?
   Call: (3) member(_0,[]) ?
   Fail: (3) member(_0,[]) ?
   Fail: (2) member(_0,[b]) ?
   Fail: (1) member(_0,[a,b]) ?

No
```