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

Association for Automated Reasoning

http://www.aarinc.org/

Automated theorem proving
Declarative programming
Automated verification

Association for Logic Programming

Logic Programming was born circa 1972, presaged by related work by Ted Elcock, Cordell Green, Pat Hayes and Carl Hewitt on applying theorem proving to problem solving and to question-answering systems.

It blossomed from Alan Robinson's seminal contribution, the Resolution Principle, all the way into a practical programming language with automated deduction at its core, through the vision and efforts of Alain Colmerauer and Bob Kowalski.

http://logicprogramming.org/

Ambivalent Syntax & Meta-variables

Ambivalent Syntax: Prolog permits the same name to be used both for function symbols and for predicate symbols, even of different arities, which is in contrast to first-order logic.
Meta-variables: Prolog permits the use of variables in the positions of atoms, both in the queries and in the clause bodies.

```
?- assert(a), assert(p(a)).

Yes

?- p(X), X.

X = a ;

No
```

Higher-Order Programming 1

The ambivalent syntax and the meta-variables support higher-order programming.

Prolog provides an indirect way of using meta-variables by means of a special predicate `call` defined as follows:

```
call(X) :- X.
```

This predicate is often used to "mask" the explicit use of meta-variables, but the outcome is the same.

```
?- p(X), call(X).

X = a ;

No
```

Higher-Order Programming 2

Recall the use of the special *univ* predicate:

```
?-  G =.. [p,a,b,c].

G = p(a, b, c) ;

No
```

Higher-Order Programming 3

Consider the following higher-order program `map(P,Xs,Ys)` where the list `Ys` is the result of applying `P` elementwise to the list `Xs`.

```
map(_,[],[]).
map(P,[X|Xs],[Y|Ys]) :- G =.. [P,X,Y], G, map(P,Xs,Ys).

square(X,Y) :- Y is X*X.

?- map(square,[1,2,3,4],R).

R = [1, 4, 9, 16] ;

No
```

Meta-Programming 1

Normally ground facts are added to the clause database:

```
?- asserta(p(a)).

Yes

?- p(X).

X = a ;

No
```

But all kinds of clauses can be added.

When a file is loaded the predicates in the file are added to the clause database as well.

Meta-Programming 2

Special predicate `clause(+Head,?Body)` which succeeds when `Head` can be unified with a clause head and `Body` can be unified with the corresponding clause body.

Gives alternative clauses on backtracking.

For facts `Body` is unified with the atom `true` — for example:

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

Here `clause` uses the equivalent:

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

Meta-Programming 3

```
?- clause(member(X,L),G).

L = [X|_]
G = true ;

L = [_|_0]
G = member(X, _0) ;

No
```

Using `clause` one can construct for example a Prolog interpreter written in Prolog, that is, a meta-interpreter.

Note that SWI-Prolog does not add the built-in predicates like `true` to the clause database.

Meta-Programming 4

The “Vanilla” meta-interpreter:

```
solve(true) :- !.
solve((A,B)) :- !, solve(A), solve(B).
solve(A) :- clause(A,B), solve(B).
```

Vanilla is commonly used to mean “plain” — derived from the use of vanilla extract as the most popular flavoring for ice cream.

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

Yes

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

No
```

Meta-Programming 5

Backtracking works as expected:

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

X = a ;

X = b ;

X = c ;

No

?- findall(X,solve(member(X,[a,b,c])),L).

L = [a, b, c] ;

No
```

Meta-Programming 6

A final example:

```
?- L = [a,B,c], A = [b,L], P = member, G =.. [P|A],
   solve(G).

L = [a, b, c]
B = b
A = [b, [a, b, c]]
P = member
G = member(b, [a, b, c]) ;

No
```

Constraint Programming 1

SWI-Prolog has a Constraint Logic Programming (CLP) library `bounds` which is briefly described here.

The constraints include the following predicates on integers:

```
?Expr #= ?Expr          ?Expr #\= ?Expr

?Expr #> ?Expr          ?Expr #< ?Expr

?Expr #>= ?Expr         ?Expr #=< ?Expr
```

These correspond to the operators: $=\quad \neq \quad > \quad < \quad \geq \quad \leq$

Constraint Programming 2

Consider the classical `SEND+MORE=MONEY` puzzle:

```
         S     E     N     D
+        M     O     R     E
------------------------------
   M     O     N     E     Y
```

All variables must take different values in the interval 0–9 and the three numbers in the equation must be well-formed.

Constraint Programming 3

```
:- ensure_loaded(library(bounds)).

puzzle([[S,E,N,D],[M,O,R,E],[M,O,N,E,Y]])  :-
  Digits  = [S,E,N,D,M,O,R,Y],
  Carries = [C1,C2,C3,C4],
  Digits  in 0..9,
  Carries in 0..1,

  M            #=          C4,
  O + 10 * C4  #=  M + S + C3,
  N + 10 * C3  #=  O + E + C2,
  E + 10 * C2  #=  R + N + C1,
  Y + 10 * C1  #=  E + D,

  M #>= 1, S #>= 1, all_different(Digits), label(Digits).
```

Constraint Programming 4

```
?- puzzle(X).

X = [[9, 5, 6, 7], [1, 0, 8, 5], [1, 0, 6, 5, 2]] ;

No
```

If the predicate `all_different` is not used then the solution is not unique.

The predicate `label` will try to assign values to the variables.

A Hilbert System: Axioms for Propositional Logic

$$A \longrightarrow B \longrightarrow A$$

$$(A \longrightarrow B \longrightarrow C) \longrightarrow (A \longrightarrow B) \longrightarrow A \longrightarrow C$$

$$(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow A \lor B \longrightarrow C$$

$$A \longrightarrow A \lor B$$

$$B \longrightarrow A \lor B$$

$$A \land B \longrightarrow A$$

$$A \land B \longrightarrow B$$

$$A \longrightarrow B \longrightarrow A \land B$$

$$((A \longrightarrow \mathtt{False}) \longrightarrow \mathtt{False}) \longrightarrow A$$

Formal Proof in Isabelle

```
theorem
  ‹A ⟶ B ⟶ A›
  ‹(A ⟶ B ⟶ C) ⟶ (A ⟶ B) ⟶ A ⟶ C›
  ‹(A ⟶ C) ⟶ (B ⟶ C) ⟶ A ∨ B ⟶ C›
  ‹A ⟶ A ∨ B›
  ‹B ⟶ A ∨ B›
  ‹A ∧ B ⟶ A›
  ‹A ∧ B ⟶ B›
  ‹A ⟶ B ⟶ A ∧ B›
  ‹((A ⟶ False) ⟶ False) ⟶ A›
  by simp_all
```

Formalization of Propositional Logic in Isabelle

```
datatype form =
  Falsity | Pro string | Imp form form | Dis form form | Con form form

primrec semantics :: ‹(string ⇒ bool) ⇒ form ⇒ bool› where
  ‹semantics _ Falsity = False› |
  ‹semantics i (Pro s) = i s› |
  ‹semantics i (Imp p q) = (if semantics i p then semantics i q else True)› |
  ‹semantics i (Dis p q) = (if semantics i p then True else semantics i q)› |
  ‹semantics i (Con p q) = (if semantics i p then semantics i q else False)›

lemma ‹semantics i (Imp p p)›
  by simp
```

```
inductive OK :: ‹form ⇒ bool› where
  ‹OK (Imp p (Imp q p))› |
  ‹OK (Imp (Imp p (Imp q r)) (Imp (Imp p q) (Imp p r)))› |
  ‹OK (Imp (Imp p r) (Imp (Imp q r) (Imp (Dis p q) r)))› |
  ‹OK (Imp p (Dis p q))› |
  ‹OK (Imp q (Dis p q))› |
  ‹OK (Imp (Con p q) p)› |
  ‹OK (Imp (Con p q) q)› |
  ‹OK (Imp p (Imp q (Con p q)))› |
  ‹OK (Imp (Imp (Imp p Falsity) Falsity) p)› |
  ‹OK p ⟹ OK (Imp p q) ⟹ OK q›

theorem soundness: ‹OK p ⟹ semantics i p›
  by (induct rule: OK.induct) simp_all
```

Propositional Logic: Soundness & Completeness

```
theorem main: ‹(∀i. semantics i p) ⟷ OK p›
```

A formal proof in Isabelle is available

(about 1000 lines including other results)

Based on "Propositional Proof Systems" by Julius Michaelis and Tobias Nipkow

(Archive of Formal Proofs 2017)