# 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**

# Prelude – Money, Money, Money

# Prelude – Money, Money, Money

**Gernot Heiser**

**Chief Research Scientist, Trustworthy Systems, Data61, CSIRO**

**Scientia Professor and John Lions Chair, UNSW**

**Gernot has built an active operating systems research group, known for… the L4-embedded microkernel that shipped in billions of mobile devices (including all recent iOS devices)**

**https://ts.data61.csiro.au/people/?cn=Gernot+Heiser**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable.**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable. In fact they are with present software technology, which is fundamentally broken.**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable. In fact they are with present software technology, which is fundamentally broken.**

**However, it is possible to architect and implement systems that are extremely cyber-robust, and the leading work is happening right here in the Trustworthy Systems Group at Data61 and UNSW.**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable. In fact they are with present software technology, which is fundamentally broken.**

**However, it is possible to architect and implement systems that are extremely cyber-robust, and the leading work is happening right here in the Trustworthy Systems Group at Data61 and UNSW.**

**In this talk I will give an overview of the technology we have developed and are continuing to develop.**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable. In fact they are with present software technology, which is fundamentally broken.**

**However, it is possible to architect and implement systems that are extremely cyber-robust, and the leading work is happening right here in the Trustworthy Systems Group at Data61 and UNSW.**

**In this talk I will give an overview of the technology we have developed and are continuing to develop. The foundation of our approach is combined with architectural approaches that allow the construction of high-assurance systems, even if the majority of the code consists of untrustworthy legacy components…**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

**Prof. Gernot Heiser (CSE UNSW): Making Systems Cyber-Secure**

**Present main-stream systems are highly vulnerable to cyber attacks, and the sentiment is widespread that these vulnerabilities are inevitable. In fact they are with present software technology, which is fundamentally broken.**

**However, it is possible to architect and implement systems that are extremely cyber-robust, and the leading work is happening right here in the Trustworthy Systems Group at Data61 and UNSW.**

**In this talk I will give an overview of the technology we have developed and are continuing to develop. The foundation of our approach is the formally verified seL4 microkernel, combined with architectural approaches that allow the construction of high-assurance systems, even if the majority of the code consists of untrustworthy legacy components…**

**27 April 2017 https://my.cse.unsw.edu.au/seminars/index.php**

# The Isabelle Proof Assistant / Theorem Prover

**“The seL4 verification uses formal mathematical proof in the theorem prover Isabelle/HOL. This theorem prover is interactive, but offers a comparatively high degree of automation. It also offers a very high degree of assurance that the resulting proof is correct.”**

**http://sel4.systems**

**http://isabelle.systems**

## Both systems are open source software :-)

# Welcome

## Teacher: Jørgen Villadsen

**Teaching Assistants: Thanks in Advance**

In short the aim of the course is to enable you to:

- Think like a logician
- Enjoy the Prolog language

# Agenda — Week #1

CampusNet Messages — Install SWI-Prolog

- Check Course Plan (Assignments & Exam)
- Obtain Textbook & Print Prolog Note (Tomorrow)
- Later Sample-Exam-1/2/3 & Skip ...-Solutions

Course Prerequisites

- Programming in Java and/or C and/or ...
- Truth Tables — REPETITION TODAY :-)
- Recursion, lists and trees (sorting algorithms are also used)

Tutorial on Logical Systems and Logic Programming (Prolog)

- Simple Rules
- Arithmetic
- Recursive Rules

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

(one plus one equals two) and (Earth is farther from the sun than Mars)

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

(one plus one equals two) and (Earth is farther from the sun than Mars)

Take negation (not) $\neg A$ and conjunction (and) $A \wedge B$ as a start

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

(one plus one equals two) and (Earth is farther from the sun than Mars)

Take negation (not) $\neg A$ and conjunction (and) $A \wedge B$ as a start

Define disjunction (or) $A \vee B$ as $\neg(\neg A \wedge \neg B)$

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

(one plus one equals two) and (Earth is farther from the sun than Mars)

Take negation (not) $\neg A$ and conjunction (and) $A \wedge B$ as a start

Define disjunction (or) $A \vee B$ as $\neg(\neg A \wedge \neg B)$

Define implication (implies) $A \rightarrow B$ as $\neg A \vee B$

# A Logical System (Propositional Logic)

The statements 'one plus one equals two' and 'Earth is farther from the sun than Venus' are both true statements; therefore, by definition, so is the following statement:

(one plus one equals two) and (Earth is farther from the sun than Venus)

Since 'Earth is farther from the sun than Mars' is a false statement, so is:

(one plus one equals two) and (Earth is farther from the sun than Mars)

Take negation (not) $\neg A$ and conjunction (and) $A \wedge B$ as a start

Define disjunction (or) $A \vee B$ as $\neg(\neg A \wedge \neg B)$

Define implication (implies) $A \rightarrow B$ as $\neg A \vee B$

Optionally define xor (exclusive or) $A \oplus B$ as $(A \vee B) \wedge \neg(A \wedge B)$

# Truth Tables

| $\neg A$ |
|---|
| 1 0 |
| 0 1 |

| $A$ | $\wedge$ | $B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 0 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $A$ | $\vee$ | $B$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 1 | 0 |
| 1 | 1 | 1 |

| $A$ | $\rightarrow$ | $B$ |
|---|---|---|
| 0 | 1 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

$A \vee B$ abbreviates $\neg(\neg A \wedge \neg B)$ and $A \rightarrow B$ abbreviates $\neg A \vee B$

Note that the symbol $\rightarrow$ is often used instead of the symbol $\Rightarrow$ in mathematical logic

# SWI-Prolog

## From Wikipedia, the free encyclopedia

# SWI-Prolog

## From Wikipedia, the free encyclopedia

SWI-Prolog is an open source implementation of the programming language Prolog, commonly used for teaching and semantic web applications. It has a rich set of features, libraries for constraint logic programming, multithreading, unit testing, GUI, interfacing to Java, ODBC and others, literate programming, a web server, SGML, RDF, RDFS, developer tools (including an IDE with a GUI debugger and GUI profiler), and extensive documentation.

# SWI-Prolog

## From Wikipedia, the free encyclopedia

SWI-Prolog is an open source implementation of the programming language Prolog, commonly used for teaching and semantic web applications. It has a rich set of features, libraries for constraint logic programming, multithreading, unit testing, GUI, interfacing to Java, ODBC and others, literate programming, a web server, SGML, RDF, RDFS, developer tools (including an IDE with a GUI debugger and GUI profiler), and extensive documentation.

SWI-Prolog runs on Unix, Windows, and Macintosh platforms.

# SWI-Prolog

## From Wikipedia, the free encyclopedia

SWI-Prolog is an open source implementation of the programming language Prolog, commonly used for teaching and semantic web applications. It has a rich set of features, libraries for constraint logic programming, multithreading, unit testing, GUI, interfacing to Java, ODBC and others, literate programming, a web server, SGML, RDF, RDFS, developer tools (including an IDE with a GUI debugger and GUI profiler), and extensive documentation.

SWI-Prolog runs on Unix, Windows, and Macintosh platforms.

SWI-Prolog has been under continuous development since 1987. Its main author is Jan Wielemaker. The name SWI is derived from Sociaal-Wetenschappelijke Informatica ("Social Science Informatics"), the former name of the group at the University of Amsterdam where Wielemaker is employed. The name of this group has changed to HCS (Human-Computer Studies).

# Logic Programming (Prolog)

Prolog variables must start with an uppercase letter:

```
likes(sam,Food) :- indian(Food), mild(Food).
likes(sam,Food) :- chinese(Food).
likes(sam,Food) :- italian(Food).
likes(sam,chips).

indian(curry). indian(dahl). indian(tandoori).
indian(kurma).

mild(dahl). mild(tandoori). mild(kurma).

chinese(chow_mein). chinese(chop_suey).
chinese(sweet_and_sour).

italian(pizza). italian(spaghetti).
```

# Testing Solutions

Load the program and enter a query after the prompt: `?-`

```
?- likes(sam,kurma).
```

```
Yes
```

Instead of `Yes` the recent versions of SWI-Prolog use `true.`

```
?- likes(sam,curry).
```

```
No
```

Instead of `No` the recent versions of SWI-Prolog use `false.`

But in this course `Yes`/`No` is still used :-)

# Computing Solutions

Enter a semicolon for each additional solution:

```
?- likes(sam,X).

X = dahl ;
X = tandoori ;
X = kurma ;
X = chow_mein ;
X = chop_suey ;
X = sweet_and_sour ;
X = pizza ;
X = spaghetti ;
X = chips ;

No
```

Here the recent versions of SWI-Prolog just omit the final step...

# Appendix — To trace Or notrace

It is optional to use the tracer – more information later...

```
?- trace, likes(P,X).

   Call: (1) likes(_1,_2) ?
   Call: (2) indian(_2) ?
   Exit: (2) indian(curry) ?
   Call: (2) mild(curry) ?
   Fail: (2) mild(curry) ?
   Redo: (2) indian(_2) ?
   Exit: (2) indian(dahl) ?
   Call: (2) mild(dahl) ?
   Exit: (2) mild(dahl) ?
   Exit: (1) likes(sam,dahl) ?

P = sam
X = dahl ;
```

# Appendix — To trace Or notrace

```
   Redo: (2) indian(_2) ?
   Exit: (2) indian(tandoori) ?
   Call: (2) mild(tandoori) ?
   Exit: (2) mild(tandoori) ?
   Exit: (1) likes(sam,tandoori) ?

P = sam
X = tandoori ;

   Redo: (2) indian(_2) ?
   Exit: (2) indian(kurma) ?
   Call: (2) mild(kurma) ?
   Exit: (2) mild(kurma) ?
   Exit: (1) likes(sam,kurma) ?

P = sam
X = kurma ;
```

# Appendix — To trace Or notrace

```
   Redo: (1) likes(_1,_2) ?
   Call: (2) chinese(_2) ?
   Exit: (2) chinese(chow_mein) ?
   Exit: (1) likes(sam,chow_mein) ?

P = sam
X = chow_mein ;

   Redo: (2) chinese(_2) ?
   Exit: (2) chinese(chop_suey) ?
   Exit: (1) likes(sam,chop_suey) ?

P = sam
X = chop_suey ;
```

# Appendix — To trace Or notrace

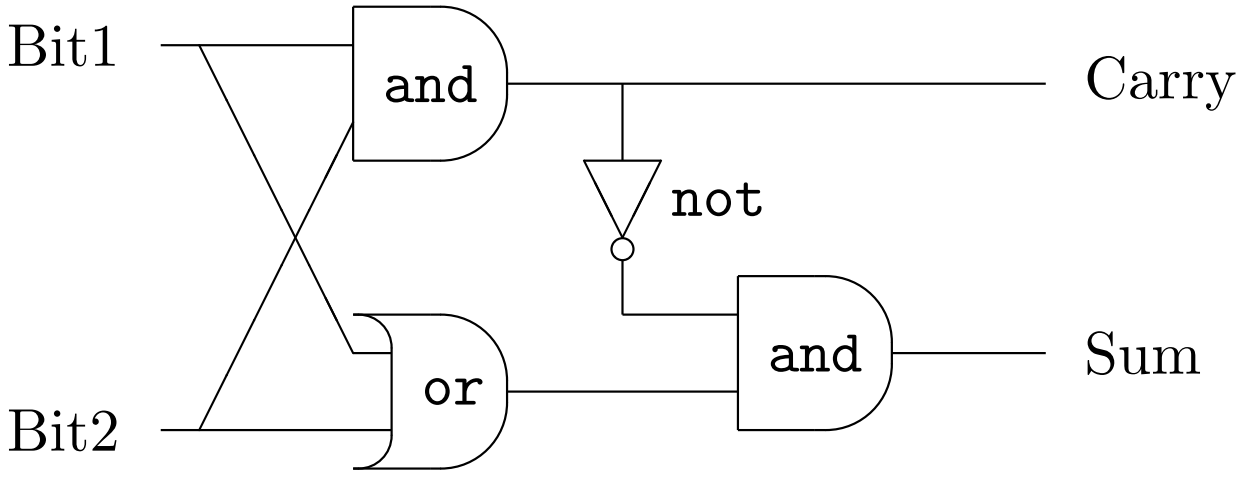
```
   Redo: (2) chinese(_2) ?
   Exit: (2) chinese(sweet_and_sour) ?
   Exit: (1) likes(sam,sweet_and_sour) ?

P = sam
X = sweet_and_sour ;

   Redo: (1) likes(_1,_2) ?
   Call: (2) italian(_2) ?
   Exit: (2) italian(pizza) ?
   Exit: (1) likes(sam,pizza) ?

P = sam
X = pizza ;
```

# Appendix — To trace Or notrace

```
   Redo: (2) italian(_2) ?
   Exit: (2) italian(spaghetti) ?
   Exit: (1) likes(sam,spaghetti) ?

P = sam
X = spaghetti ;

   Redo: (1) likes(_1,_2) ?
   Exit: (1) likes(sam,chips) ?

P = sam
X = chips ;

No
```

In SWI-Prolog the trace look a little bit different... :-)

# Simple Rules




$$\text{Sum} = \neg(\text{Bit1} \wedge \text{Bit2}) \wedge (\text{Bit1} \vee \text{Bit2})$$

$$\text{Carry} = \text{Bit1} \wedge \text{Bit2}$$

```
?- halfadder(0,1,1,0).

Yes
```

```
?- halfadder(1,1,0,0).

No
```

# Query

```
?- halfadder(1,1,Sum,Carry).

Sum = 0
Carry = 1

Yes
```

Recall:

$$\text{Sum} \;=\; \neg(\text{Bit1} \wedge \text{Bit2}) \wedge (\text{Bit1} \vee \text{Bit2})$$
$$\text{Carry} \;=\; \text{Bit1} \wedge \text{Bit2}$$

# Program

```
not(0,1).
not(1,0).

and(0,0,0).
and(0,1,0).
and(1,0,0).
and(1,1,1).

or(0,0,0).
or(0,1,1).
or(1,0,1).
or(1,1,1).
```

# Program Continued

```
halfadder(Bit1,Bit2,Sum,Carry) :-
  and(Bit1,Bit2,Carry),
  not(Carry,X),
  or(Bit1,Bit2,Y),
  and(X,Y,Sum).
```

Recall:

$$\text{Sum} = \neg(\text{Bit1} \wedge \text{Bit2}) \wedge (\text{Bit1} \vee \text{Bit2})$$

$$\text{Carry} = \text{Bit1} \wedge \text{Bit2}$$

# Another Query

Using variables for all 4 arguments:

```
?- halfadder(Bit1,Bit2,Sum,Carry).

Bit1 = 0
Bit2 = 0
Sum = 0
Carry = 0 ;

Bit1 = 0
Bit2 = 1
Sum = 1
Carry = 0 ;
```

# Another Query Continued

```
Bit1 = 1
Bit2 = 0
Sum = 1
Carry = 0 ;

Bit1 = 1
Bit2 = 1
Sum = 0
Carry = 1 ;

No
```

Hence 4 solutions as expected!

# Arithmetic

```
?- Z is 2+2.

Z = 4

Yes

?- Z = 2+2.

Z = 2+2

Yes

?- Z = 2+2, Z = 4.

No
```

# Arithmetic — Exponentiation

```
?- Z is 2**10.

Z = 1024

Yes
```

Note:

$2^{10} = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 1024$

$2^{1} = 2$

$2^{0} = 1$

# Query

Examples not using the built-in arithmetic (the ** operator):

```
?- power(2,10,Z).

Z = 1024

Yes

?- power(2,10,1024).

Yes

?- power(2,10,1000).

No
```

# Program

```
power(_,0,1).
power(X,Y,Z) :-
  Y > 0,
  Y1 is Y-1,
  power(X,Y1,Z1),
  Z is X*Z1.
```

No loops but recursion in Prolog...!

No type system but so-called instantiation patterns can be added:

```
% power(+Integer1,+Integer2,?Integer3).
```

```
% halfadder(?Bit1,?Bit2,?Sum,?Carry).
```

But these lines are just comments for the programmers...!

# Appendix — To trace Or notrace

```
?- trace, power(10,2,Z).

   Call: (1) power(10,2,_1) ?
   Call: (2) 2 > 0 ?
   Exit: (2) 2 > 0 ?
   Call: (2) _2 is 2-1 ?
   Exit: (2) 1 is 2-1 ?
   Call: (2) power(10,1,_3) ?
   Call: (3) 1 > 0 ?
   Exit: (3) 1 > 0 ?
   Call: (3) _4 is 1-1 ?
   Exit: (3) 0 is 1-1 ?
   Call: (3) power(10,0,_5) ?
   Exit: (3) power(10,0,1) ?
   Call: (3) _6 is 10*1 ?
   Exit: (3) 10 is 10*1 ?
   Exit: (2) power(10,1,10) ?
   Call: (2) _1 is 10*10 ?
   Exit: (2) 100 is 10*10 ?
   Exit: (1) power(10,2,100) ?
```

# Appendix — To trace Or notrace

```
Z = 100

Yes
```

One can type `notrace` to disable the tracer — but this leaves the so-called debugger enabled — so better type `nodebug` and both the tracer and the debugger are disabled.

# Program & Query

```
parent(alan,bill).
parent(bill,carl).
parent(carl,dean).
parent(dean,eric).

grandparent(X,Y) :- parent(X,Z), parent(Z,Y).



?- grandparent(alan,carl).

Yes
```

# More...

```
?- grandparent(X,carl).

X = alan

Yes

?- grandparent(alan,Y).

Y = carl

Yes
```

# Final Query

```
?- grandparent(X,Y).

X = alan
Y = carl ;

X = bill
Y = dean ;

X = carl
Y = eric ;

No
```

# Parent-Ancestor — How NOT to Do It

Grandparent handles 2 generations:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

The following approach to a general ancestor definition can handle only a given number of generations:

```
ancestor(X,Y) :-
  parent(X,Y).
ancestor(X,Y) :-
  parent(X,Z), parent(Z,Y).
ancestor(X,Y) :-
  parent(X,Z), parent(Z,V), parent(V,Y).
ancestor(X,Y) :-
  parent(X,Z), parent(Z,V), parent(V,W), parent(W,Y).
% ...
```

# Parent-Ancestor I

```
?- ancestor(alan,eric).

Yes

?- ancestor(alan,Y).

Y = bill ;

Y = carl ;

Y = dean ;

Y = eric ;

No
```

# Parent-Ancestor II

```
?- ancestor(X,Y).

X = alan
Y = bill ;

X = bill
Y = carl ;

X = carl
Y = dean ;

X = dean
Y = eric ;

X = alan
Y = carl ;
```

# Parent-Ancestor III

```
X = alan
Y = dean ;

X = alan
Y = eric ;

X = bill
Y = dean ;

X = bill
Y = eric ;

X = carl
Y = eric ;

No
```

# A Tricky One

```
?- ancestor(X,eric).

X = dean ;

X = alan ;

X = bill ;

X = carl ;

No
```

Compare the order of the solutions with the previous query!

# Recursive Rules

Grandparent handles 2 generations:

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

The following approach to a general ancestor definition can handle an arbitrary number of generations:

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Again: No loops but recursion in Prolog...!

# Today's Final Slide

Next week: Prolog Basics — A bit more systematic approach...

# Today's Final Slide

Next week: Prolog Basics — A bit more systematic approach...

NOW:

## Solve today's exercises in the databars

The programs on today's slides are available in CampusNet :-)

Solutions will be available in CampusNet!