

# CSC4504/Prolog. : Formal Languages & Applications

**J Paul Gibson, D311**

paul.gibson@telecom-sudparis.eu

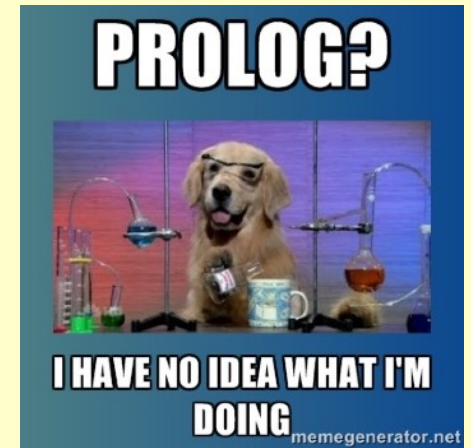
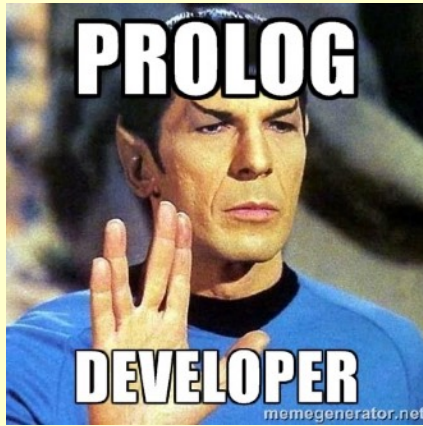
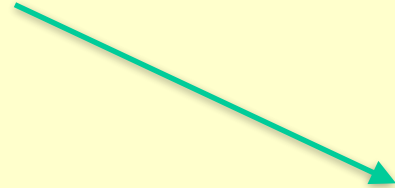
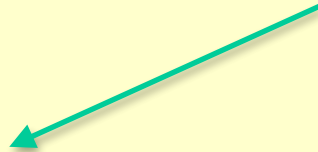
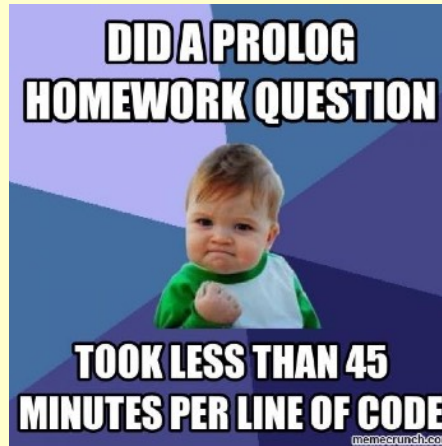
<http://www-public.telecom-sudparis.eu/~gibson/Teaching/CSC4504/>

An Introduction To **Prolog**

[/~gibson/Teaching/CSC4504/Problem8-Prolog.pdf](http://www-public.telecom-sudparis.eu/~gibson/Teaching/CSC4504/Problem8-Prolog.pdf)

*How many Prolog programmers does it take to change a lightbulb?*

*false.*



# Logic programming with Prolog

We are going to look at logic programming

We choose Prolog as a typical example

- Started around 1970

- Used for many applications :

relational databases, mathematical logic, abstract problem solving, understanding natural language, design automation, symbolic equation solving, biochemical structure analysis, ...

- Most recent use is in AI

# Logic programming with Prolog: history

Prolog from research at the University of Aix-Marseille in the late 60's and early 70's.

Alain **Colmerauer** and Phillipe **Roussel** (University of Aix-Marseille) collaborated with Robert **Kowalski** (University of Edinburgh) to create the design of Prolog as we know it today.

- Kowalski - the theoretical framework
- Colmerauer's -formalize the Prolog language.

1972 birthdate of Prolog.

The first Prolog compiler - David **Warren**, an expert on Artificial Intelligence at the University of Edinburgh – *Warren's Abstract Machine (1983)*

# Logic programming with Prolog: some reading

*Predicate Logic as a programming language*, Kowalski, 1974

*Prolog - The language and its implementation compared with LISP*, Warren, Pereira and Pereira, 1977

*The Early Years Of Logic Programing*, Robert Kowalski, 1988

*The birth of Prolog*, Colmeraurer and Roussel, 1996

# For beginners

Writing a Prolog program is not like specifying an algorithm in the conventional way.

The programmer asks what formal relationships and objects occur in the problem.

Program searches for what relationships are ‘true’ in the desired solution.

Prolog is descriptive (what) rather than prescriptive (how)

# Prolog Computation

The way the computer carries out a computation for Prolog is specified partly by:

- the logical declarative semantics of Prolog
- the new facts Prolog can ‘infer’ from the given facts
- explicit control information given in the program

# Prolog Overview

Prolog is a practical and efficient implementation of many aspects of ‘intelligent’ program execution like:

- non-determinism,
- parallelism,
- pattern-directed procedure call

Prolog provides a uniform data structure called the *term*, around which all data and programs are constructed.

A Prolog program consists of a set of clauses, where each *clause* is either:

- a fact about the given information ,or
- a rule about how the solution may relate to or be inferred from the given facts



# Introducing Prolog

To start, we look at essential elements of the language in real programs, without becoming diverted by details, formal rules, and exceptions.

We aren't trying to be complete or precise

We hope to be writing programs ASAP

Concentrate on the basics:

- facts,
- questions,
- variables,
- conjunctions,
- rules

# Objects and relationships

Prolog is used when we wish the computer to solve problems that can be expressed in the form of objects and their relationships.

Example:

*Paul owns the car*

This declares that a relationship (ownership) exists between one object (Paul) and another object (the car).

Note:

- the relationship has an order
- the objects are concrete
- we didn't say 'Paul owns a car'!!

# Abstraction in relationships

Some relationships don't always mention all the objects that are involved.

Example:

*Paul is intelligent.*

Here, there is a relationship (being intelligent) which involves Paul. We do not say who finds Paul intelligent (or why)

We abstract away from this information as we choose only to say what we want to say.

In Prolog, the amount of detail given has influence on the things you can accomplish

# A bit of philosophy

We are all familiar with rules like:

*2 people are sisters if they are both female and have the same parents*

This tells us:

- something about what it means to be sisters
- how to find out if 2 people are sisters

Note:

- these rules are usually oversimplified, but acceptable as definitions.
- do not expect a definition to tell us everything about something

# Philosophy continued ...

Most people would agree that there is ‘more to being sisters’ in real life than the rule would suggest.

When solving a problem, we must concentrate only on the rules which can help us.

We should consider an imaginary and simplified definition if it is sufficient for our purposes.

Prolog programming consists of:

- declaring *facts* about objects and relationships
- declaring *rules* about objects and relationships
- asking *questions* about objects and relationships

# The philosophy of sisters

Suppose we told Prolog a rule about sisters. We could then ask whether Paul and Peter were sisters. Prolog would search through everything it knows (about sisters, paul and peter) and return **yes** or **no**.

## **Question:**

if Prolog does not have enough information to decide if Paul and Peter are sisters then what should it do?

**Answer:** this defines part of the boundary between philosophy and logic ... we shall see what Prolog actually does (later on)

**Further reading:** closed world assumption vs open world assumption

# The syntax of facts

Informally we know that Paul likes Beer.

Formally, in Prolog, we write: **likes(paul, beer).**

Note:

- names of objects and relationships must begin with a lower-case
- the relationship is written first
- the objects are separated by commas
- objects are enclosed by round brackets
- the full stop character ‘.’ must come at the end of a fact

# Some example facts

<b>good(beer).</b>	<i>Beer is good</i>
<b>drinks(paul, beer).</b>	<i>paul drinks beer</i>
<b>buys(paul, beer, peter).</b>	<i>paul buys beer for peter</i>
<b>owes(peter, beer, paul).</b>	<i>peter owes paul beer</i>
<b>mix(beer,vodka,drunk).</b>	<i>mixing beer and vodka makes you drunk</i>

Note: when using names we must decide how to interpret the name. The programmer decides on the interpretation... so make it consistent and comprehensible.



# Some terminology

The names of objects in round brackets are arguments

The name of the relationship before the round brackets is the predicate

The names chosen are arbitrary, but we normally select names to correspond to their interpretation.

The order of arguments is also arbitrary, but again we should try to match intuition in a consistent manner.

Relationships can have an arbitrary number of arguments

More complex relationships require many arguments:

**cocktail(paul, favourite, beer, gin, wine, whiskey).**

A collection of facts is called a *database*

# Questions in Prolog

Once we have some facts, we can ask questions about them.

In Prolog, a question looks like a fact, except that we put a special symbol before it ... a question mark and hyphen.

**?- likes(paul,beer).**

When a question is asked, Prolog searches through the database of facts looking for matches.

Two facts match if their predicates are the same and if their arguments are the same.

If a matching fact is found then Prolog responds **yes**  
otherwise Prolog responds **no**

# Facts and questions

Consider the following database of facts:

**likes(paul,beer).**

**likes(paul,wine).**

**canmakeyoudrunk(wine).**

**canmakeyoudrunk(beer).**

**canmakeyoudrunk(paul).**

Now we pose the following questions:

**likes(paul,money)**

**isUniversity(oxford)**

In both cases, the answer is **no**, even if the intended meaning is true. In prolog **no** is returned when *nothing matches*


# Installing SWI Prolog <http://www.swi-prolog.org/download/stable>









## Download SWI-Prolog stable versions

[Home](#)[DOWNLOAD](#)[DOCUMENTATION](#)[TUTORIALS](#)[COMMUNITY](#)[USERS](#)[WIKI](#)

 Linux versions are often available as a package for your distribution. We collect information about available packages and issues for building on specific distros [here](#). We provide a [PPA](#) for [Ubuntu](#)

 Please check the [windows release notes](#) (also in the SWI-Prolog startup menu of your installed version) for details.

 Examine the [ChangeLog](#).

Binaries		
	21,163,178 bytes	<a href="#">SWI-Prolog 7.6.3 for Microsoft Windows (64 bit)</a> Self-installing executable for Microsoft's Windows 64-bit editions. Requires at least Windows 7. See the <a href="#">reference manual</a> for deciding on whether to use the 32- or 64-bits version. This binary is linked against GMP 6.1.1 which is covered by the LGPL license. <b>SHA256:</b> 6d110b72dc092c829c6a82418e11752483f7360ce01398bfc1360addf1f4fb46
	20,013,203 bytes	<a href="#">SWI-Prolog 7.6.3 for Microsoft Windows (32 bit)</a> Self-installing executable for MS-Windows. Requires at least Windows 7. Installs <b>swipl-win.exe</b> and <b>swipl.exe</b> . This binary is linked against GMP 6.1.1 which is covered by the LGPL license. <b>SHA256:</b> 219115834b2b743c4a68378cc122f85a906d368bad9600f4700cd992b7ef7c13
	23,845,993 bytes	<a href="#">SWI-Prolog 7.6.3 for MacOSX 10.6 (Snow Leopard) and later on intel</a> Mac OS X disk image with <b>relocatable application bundle</b> . Needs <a href="#">xquartz</a> (X11) installed for running the <a href="#">development tools</a> . Currently, version 2.7.11 is required. You can check the version by opening an X11 application and then checking 'about' in the X11 menu. The bundle also provides the commandline tools in <code>Contents/MacOS</code> . The command line tools need at least MacOS <b>10.6</b> (Snow Leopard). The graphical application needs at least MacOS <b>10.7</b> (Lion). <b>SHA256:</b> e4c20d3e8d22bd2f9dab1287cd1c1d8a102e1bcb01985909d9a03646573ba413
Sources		
	16,772,590 bytes	<a href="#">SWI-Prolog source for 7.6.3</a> Sources in <code>.tar.gz</code> format, including packages and generated documentation files. See <a href="#">build instructions</a> . <b>SHA256:</b> 9c0be513dc98e6ca420d109f7a455a6593840682cf1b0e21876d1e71e7f35d39
Documentation		
	2,423,184 bytes	<a href="#">SWI-Prolog 7.6.3 reference manual in PDF</a> SWI-Prolog reference manual as PDF file. This does <i>not</i> include the <a href="#">package</a> documentation.

[Show all files](#)

# PDT Prolog IDE for Eclipse (For later ...)

## The Prolog Development Tool – A Prolog IDE for Eclipse

The PDT is a Prolog IDE provided as a plug-in for the Eclipse Platform. All PDT features are implemented for SWI-Prolog, most also for Logtalk. All native SWI-Prolog development tools (graphical tracer / debugger, profiler, ...) can be used within the PDT. If you want to use the PDT with other Prolog implementations, [read this](#).



Since version 1.0 the PDT is based on **consulted code**. All its elements (navigator, editor, outline, search, context view, code completion, etc.) show you the state of the code loaded in the Prolog process currently displayed in the Prolog Console. It is recommended to have an [entry point file](#) for each project, which you consult when you start working on that project. The PDT helps you in [creating entry point files](#).

- Click on any section title below to see the related manual section
- Click [here](#) for full manual on one page (for quick printing)

### Project Explorer

- Show source files and quick load files (QLF)
- Show edited external files (blue link symbol)
- Highlighting of entry points (yellow arrow)
- Highlighting of consulted files (green background)

### Prolog Editor

- Syntax highlighting
- Singleton variable highlighting
- Code Completion (module-aware)
- Outline (module- and multifile-aware)
- Quick outline displays predicate documentation (Ctrl O)
- Warning and errors annotations (upon saving)
- Breakpoints for the debugger
- Keyboard shortcuts

### Prolog Console

- Interaction with multiple Prolog processes
- History for each Prolog process
- Code Completion
- Errors and warnings hyperlinked to editors
- Color highlighting of warnings and errors
- Adjustable fonts and colors
- Automated creation of load files
- Automated reconsult of loaded files upon restart
- Interaction with SWI-Prolog tools
- Keyboard shortcuts

### Prolog Search

- Find definition (module- and import-aware)
- Find references (module-, import-, metaterm- and documentation-aware)
- Global search, context search from editor, search from console.

### Context View

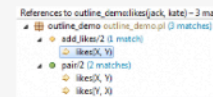
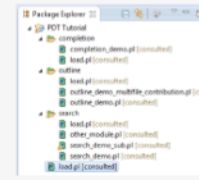
- Visualize call relations and dead code
- Visualize predicate properties (exported, dynamic, ...)
- Focus on active editor, its callers and callees

### SWI Prolog Debugger

- Source level debugger
- Breakpoints (connected to PDT editor)
- Stack state
- Variable bindings
- Backtracking visualisation

### SWI Prolog Profiler

- Statistics on time spent in invoked predicates



<http://sewiki.iai.uni-bonn.de/research/pdt/docs/start>

# Using SWI Prolog

Write a program in a text file:

```
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/syllogism.pl

/**
 * A simple example of a syllogism in Prolog
 *
 * MAJOR PREMISE - All men are mortal
 * MINOR PREMISE - Socrates is a man
 * CONCLUSION (DEDUCTION) - Socrates is mortal
 *
 **/

is_a( socrates, man ).

is_a( X, mortal ) :-
is_a( X, man ).


/*
 *is_a( socrates, mortal ).
 */
```

# Using SWI Prolog

1 ?- **consult(syllogism).**

% syllogism compiled 0.00 sec, 424 bytes

Or menu-file-consult



```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 7.3.19)
Copyright (c) 1990-2015 University of Amsterdam, VU Amsterdam
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/syllogism.pl compiled 0.00 sec, 2 clauses
1 ?- is_a(socrates, mortal).
true
```

# A first experiment: Paul and Andrew share a dad so they are siblings???

1 ?- [user]. **Interactive programming- input facts and rules**

**CTRL RETURN ??**

|: sibling(X,Y) :- parent(Z,X), parent (Z,Y).

**RULE**

ERROR: user://1:10:39: Syntax error: Operator expected

**Watch out for spaces**

|: sibling(X,Y) :- parent(Z,X), parent(Z,Y).

**RULE**

|: parent(paul, dad).

**FACT**

|: parent(andrew, dad).

**FACT**

|:

**CTRL D - back to goals/queries**

|:

% user://1 compiled 0.02 sec, 4 clauses

true.

2 ?- sibling(paul, andrew).

**QUERY**

false.

**Why is this not true?**

**Watch out for ordering of parameters**



**A first  
experiment  
: Paul and  
Andrew  
share a dad  
so they are  
siblings???  
... let's get  
it right**

```
|: sibling(X,Y) :- parent(Z,X), parent(Z,Y).
```

```
|: parent(dad,paul).
```

```
|: parent(dad,andrew).
```

```
|:
```

```
% user://1 compiled 0.01 sec, 4 clauses  
true.
```

```
2 ?- sibling(paul,andrew).
```

```
true.
```

```
3 ?- sibling(andrew,paul).
```

```
true.
```

```
4 ?- sibling(X,Y).
```

```
X = Y, Y = paul .
```

RETURN finishes the search

```
5 ?- sibling(X,Y).
```

```
X = Y, Y = paul
```

```
X = paul,
```

```
Y = andrew
```

```
X = andrew,
```

```
Y = paul
```

```
X = Y, Y = andrew.
```

; “semi-colon” keeps searching

**Write a Prolog program to capture the following properties:**

**paul is the brother of jill**

**jill is the sister of andrew**

**Brothers share a common parent**

**Sisters share a common parent**

**Siblings share a common parent**

**Check whether or not (depending on your program)**

**paul and andrew are siblings**

## Built-in predicates

**write**

```
?- write(hello).  
hello  
true.
```

```
?- write>Hello).  
_G2137  
true.
```

```
?- write(hello world).  
ERROR: Syntax error: Operator expected  
ERROR: write(hello  
ERROR: ** here **  
ERROR: world) .
```

```
?- write('hello world').  
hello world  
true.
```

## History Commands:

- !!. Repeat last query
- !nr. Repeat query numbered <nr>
- !str. Repeat last query starting with <str>
- !?str. Repeat last query holding <str>
- ^old^new. Substitute <old> into <new> of last query
- !nr^old^new. Substitute in query numbered <nr>
- !str^old^new. Substitute in query starting with <str>
- !?str^old^new. Substitute in query holding <str>
- h. Show history list
- !h. Show this list

## History Commands Examples:

17 ?- h.

```
1 [user].
2 sibling(paul,andrew).
3 sibling(andrew,paul).
4 sibling(X,Y).
5 sibling(X,Y).
6 sibling(X,Y).
7 sibling(X,Y).
8 sibling(X,Y).
9 write(hello).
10 write>Hello).
11 ?- write('hello world').
12 write('hello world').
13 write("hello world").
14 write("hello world").
15 write('hello world').
16 sibling(X,Y).
```

18 ?- !10.

```
write>Hello).
```

```
_G3436
```

```
true.
```

19 ?- !sibling.

```
sibling(X,Y).
```

```
X = Y, Y = paul .
```

**Terminate the interaction with  
halt.**

# Loading an existing prolog database (prolog-example1.pl)

```
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/prolog-example1.pl
```

```
/**
```

```
* A simple first example
```

```
*
```

```
**/
```

```
likes(paul,beer).
```

```
likes(paul,whiskey).
```

```
likes(andrew,beer).
```

```
likes(andrew,wine).
```

# Listing an existing prolog database



## List the program, predicates or clauses

[Home](#)[DOWNLOAD](#)[DOCUMENTATION](#)[TUTORIALS](#)[COMMUNITY](#)[USERS](#)[WIKI](#)[Documentation](#)[Reference manual](#)[Built-in Predicates](#)[Notation of Predicate De](#)[Character representatio](#)[Loading Prolog source fi](#)[Editor Interface](#)[List the program, predi](#)[listing/1](#)[listing/0](#)[portray\\_clause/1](#)[portray\\_clause/2](#)[Verify Type of a Term](#)[Comparison and Unifica](#)[Control Predicates](#)[Meta-Call Predicates](#)[Delimited continuations](#)[Exception handling](#)[Handling signals](#)[DCG Grammar rules](#)[Database](#)[Declaring predicate pro](#)[Examining the program](#)[Input and output](#)[Status of streams](#)[Primitive character I/O](#)

### 4.5 List the program, predicates or clauses

#### **listing(*:Pred*)**

List predicates specified by *Pred*. *Pred* may be a predicate name (atom), which lists all predicates with this name, regardless of their arity. It can also be a predicate indicator (*<name>/<arity>* or *<name>/|<arity>*), possibly qualified with a module. For example: `?- listing(lists:member/2) ..`

A listing is produced by enumerating the clauses of the predicate using [clause/2](#) and printing each clause using [portray\\_clause/1](#). This implies that the variable names are generated (*A*, *B*, ...) and the layout is defined by rules in [portray\\_clause/1](#).

#### **listing**

List all predicates from the calling module using [listing/1](#). For example, `?- listing.` lists clauses in the default `user` module and `?- lists:listing.` lists the clauses in the module `lists`.

#### **portray\_clause(+Clause)**

Pretty print a clause. A clause should be specified as a term `<Head> :- <Body>!`. Facts are represented as `<Head> :- true!` or simply `<Head>`. Variables in the clause are written as *A*, *B*, ... . Singleton variables are written as `_`. See also [portray\\_clause/2](#).

#### **portray\_clause(+Stream, +Clause)**

Pretty print a clause to *Stream*. See [portray\\_clause/1](#) for details.

# Loading an existing prolog database

*Can type the command consult or use the file menu*

```
1 ?- consult('/Users/jpaulgibson/Documents/MyPrograms/MyProlog/prolog-example1.pl').  
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/prolog-example1.pl compiled 0.00 sec, 5  
clauses  
true.
```

```
2 ?- likes(X,Y).
```

```
X = paul,
```

```
Y = beer
```

```
X = paul,
```

```
Y = whiskey
```

```
X = andrew,
```

```
Y = beer
```

```
X = andrew,
```

```
Y = wine.
```

*Press space between pairs*

```
3 ?-
```

Note: to load a file into the interpreter we can also write:

```
['/Users/jpaulgibson/Documents/MyPrograms/MyProlog/prolog-example1.pl'].
```



## Prolog has built-in lists

?- [a,b,c,d] = [H|T].

H = a, T = [b,c,d]

?- [a,b,c,d] = [H1,H2|T].

H1 = a, H2 = b, T = [c,d]



# Appending lists

```
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/list-append.pl
```

```
/**  
 * A simple list append example  
 *  
 **/
```

```
append([],List,List).
```

```
append([H|Tail],X,[H|NewTail]) :- append(Tail,X,NewTail).
```

```
['/Users/jpaulgibson/Documents/MyPrograms/MyProlog/list-append.pl'].
```

```
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/list-append.pl compiled 0.00 sec, 3  
clauses  
true.
```

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

```
X = [a, b, c, d, e].
```

**Question: how does this work?**

# Prolog has built-in lists

## List unification

$[X|Y]$  unifies with  $[a,b,c]$  with the unifier  $\{X = a, Y = [b,c]\}$ .

$[X|Y]$  unifies with  $[a,b,c,d]$  with the unifier  $\{X = a, Y = [b,c,d]\}$ .

$[X|Y]$  unifies with  $[a]$  with the unifier  $\{X = a, Y = []\}$ .

$[X|Y]$  does not unify with  $[]$ .

# Prolog has built-in lists

The `append` predicate can also be used the other way round for splitting a list into two separate parts:

```
?- append(L1,L2,[a,b,c]).  
L1 = [], L2 = [a,b,c] ?;  
L1 = [a], L2 = [b,c] ?;  
L1 = [a,b], L2 = [c] ?;  
L1 = [a,b,c], L2 = [] ?;  
no
```

The predicates `member` and `append` are built into most Prolog systems; in SICStus Prolog, they are part of a library that needs to be loaded explicitly. A source program file that uses these predicates should include the following directive:

```
:- use_module(library(lists)).
```

# Prolog has built-in lists

## List unification - we can see this with the trace functionality

?- trace.

trace on

true.

[trace] 8 ?- append([a,b,c],[d,e],X).

Call: (6) append([a, b, c], [d, e], \_G3547) ? creep

Call: (7) append([b, c], [d, e], \_G3629) ? creep

Call: (8) append([c], [d, e], \_G3632) ? creep

Call: (9) append([], [d, e], \_G3635) ? creep

Exit: (9) append([], [d, e], [d, e]) ? creep

Exit: (8) append([c], [d, e], [c, d, e]) ? creep

Exit: (7) append([b, c], [d, e], [b, c, d, e]) ? creep

Exit: (6) append([a, b, c], [d, e], [a, b, c, d, e]) ? creep

X = [a, b, c, d, e].

?- notrace.

trace off

# Prolog has built-in lists

## **PROBLEM : write a Prolog program for reversing a list**

```
1 ?- ['/Users/jpaulgibson/Documents/MyPrograms/MyProlog/list-reverse.pl'].  
% /Users/jpaulgibson/Documents/MyPrograms/MyProlog/list-reverse.pl compiled 0.00 sec, 3 clauses  
true.
```

```
2 ?- reverse([1,2,3], X).  
X = [3, 2, 1].
```

# How do queries (with variables) execute?

To begin, all we can get back from our database of facts is the information that we have put in.

It would be more interesting to ask things like:

*what does paul like?*

*what canmakeyoudrunk?*

In Prolog, this is what we use variables for ... these are names (starting with capital letters) which stand for objects to be determined by Prolog. A variable can be:

- *instantiated* ... when there is an object that it stands for, or
- *not instantiated* ... when what it stands for is not yet known



# Questions with variables

When faced with a question such as:

*what does Paul like?*

Prolog will search through all its facts to find a match.

**?-likes(paul,X).**

**X = beer.**

Prolog will then wait for further instructions.

**Question:** why beer and not wine?

**Question:** why not both?

# How it searches

- When the question is asked, the variable **X** is not instantiated
- Prolog searches for a fact where:
  - the predicate is likes
  - the first argument is paul
- Once a match is found, it instantiates **X** as the 2nd parameter
- Prolog searches in top-down order in the file (on the page)
- So, **likes(paul,beer)** is found first and so **X** is set to **beer**.
- **Note:** Prolog now marks the place where this match was found ... this is important later.

# What to do once a match is made

There are 2 logical choices as to how to continue:

- If you are satisfied with one answer you type **RETURN**
- If you want to search for more matches then type **; RETURN**

If you continue your search then:

- Prolog forgets that it has instantiated X to beer
- Continues the search at the point at which it previously found a match (the place marker)
- If it finds a match then you can again choose to continue as before

Finishing the search:

Prolog returns **no** when no more matches are found after the current place marker. When this occurs you can try another question or give more facts

# Conjunctions

We may wish to structure our questions:

*does paul like beer and wine?*

*and* is the logical conjunction ... represented in Prolog by a comma

```
?- likes(paul,beer), likes(paul,wine).
```

We can now ask more interesting questions.

**Question:** how can we find something that paul likes and which can make you drunk?

# Disjunctions (; option)

```
happy1(X) :- rich(X).  
happy1(X) :- famous(X).
```

```
happy2(X) :- rich(X) ; famous(X).
```

Combining conjunction and disjunction:

```
happy3(X) :- attractive(X), ( rich(X) ; famous(X) ).
```

# Paul likes (some)thing(s) that can make him drunk?

If we type in the following question:

**?- likes(paul,X) , makesyoudrunk(X).**

Prolog will reply

**X=beer**

How does this work?

Prolog attempts to satisfy the first goal ... and marks the place in the database where it finds the first matching fact. It then attempts to satisfy the second goal (using the previous instantiation) and marks the first matching place in the database.

**Note:** we have not shown that *Paul likes all things that can make you drunk...* even though this is true in the database and false in the real world.

# Backtracking: an introduction

The previous example illustrated a very simple case where: the initial values that variables are instantiated to do not change at later points in the search.

**Question:** can you think of an example set of facts, and a question which could not be matched in this way ... even though a match could be found if we could *go back and re-instantiate* a variable to a different value?

**Note:** this technique is known as backtracking and is fundamental to programming in Prolog.

# Backtracking: an example

<b>likes(paul,beer).</b>	<i>1</i>
<b>hates(patricia, football).</b>	<i>2</i>
<b>likes(paul,wine).</b>	<i>3</i>
<b>likes(paul, football).</b>	<i>4</i>
<b>hates(patricia, motorracing).</b>	<i>5</i>
<b>?- likes(paul, X), hates(patricia,X).</b>	<i>6</i>

## Finding another match ---

Type **;RETURN** and Prolog re-starts the search at the current markers.

## Prolog execution:

```
match1 X = beer
marker1 = 1
nomatch2
restart 2
match X = wine
marker1 = 3
nomatch 2
restart 4
match X = football
marker 1 = 4
match2
marker2= 2
X = football
```



# Rules: more complex reasoning

We wish to express the following:

*Paul likes everything that can make you drunk*

*Peter likes only those things that can make you drunk*

We do this using rules:

**makeyoudrunk(X) :- likes(peter,X).**

**likes(paul,X) :- makeyoudrunk(X)**

Terminology:

- **likes(peter,X)** is called the *body* of the rule (1)
- **likes(paul,X)** is called the *head* of the rule (2)
- **h:-b** can be interpreted as '*h if b*'

# Some More Terminology

A *clause* (of a predicate) is any fact or rule (which includes this predicate).

For example,

**likes(paul,beer).**

**likes(paul,X):- makesyoudrunk(X), cheap(X).**

are both *clauses* of the predicate **likes**

**Note:** when trying to answer a question, Prolog may not use, or need to use, all the clauses.

# Some Reasoning

```
/*1*/ drinker(paul).  
/*2*/ likes(patricia, wine).  
/*3*/ likes(patricia, chocolate).  
/*4*/ likes(paul,X) :- likes(X,wine).  
/*5*/ drinks(paul,X) :- likes(paul,X).
```

## **Note:**

the comments  
between the  
/\* ... \*/

## **Question (1):**

**?- drinks(paul,X).**

results in what response?

## **Question (2):**

which clause(es) is/are not  
relevant to question (1)

# Recursive Rules

Imagine that we wish to examine the ancestry between certain people.

We have a list of parenthood facts. For example:

**parent(tom,paul).**

**parent(paul, andrew).**

**parent(bill, tom).**

Here, we should be able to see that bill is paul's grandparent because bill is tom's parent and tom is paul's parent. We can thus write a grandparent rule:

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

# Recursive rules continued ...

**parent(tom,paul).**

**parent(paul, andrew).**

**parent(bill, tom).**

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

We can now deduce:

**grandparent(bill, paul).**

**grandparent(tom, andrew).**

But, how do we note that andrew is an ancestor of tom?

**Answer:** why not just write a **greatgrandparent** rule?

# Great grandparent rule

**ggp(X,Y) = grandparent(X,Z), parent(Z,Y).**

Now, to define ancestry we just have to say that if A is:

- a parent
- a grandparent, or
- a greatgrandparent, or ...

**Question:** can you write this rule in prolog?

of B then

A is an ancestor of B

**Question:** this will work to show that bill is an ancestor of andrew, but why is it not a good approach to ancestry?

# Ancestry by Recursive Rules

Without recursion we have to keep on adding rules for being an ancestor indefinitely.

A better way is to define ancestor in terms of itself:

A is an ancestor of B if A is B's parent or if A is an ancestor of B's parent.

**Question:** can you write this in prolog

**Hint:** it requires 2 rules ...

- a non-recursive base case
- a recursive case

# Prolog Ancestry

**ancestor(X,Y) :- parent(X,Y).**

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

Now, if we query or database:  
with **?- ancestor(bill, andrew).**

**parent(tom,paul).**  
**parent(paul, andrew).**  
**parent(bill, tom).**

Prolog will reply

**yes.**

**Question:** can you follow the prolog process to see how this answer is achieved?



# Recursive Rules and ordering

We originally gave the rules in the following order:

**ancestor(X,Y) :- parent(X,Y).**

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

In Prolog, the order of the rules is often very important with regard to the processing of queries.

## **Question:**

In this case, would it matter if we had reversed the ordering?

# Anonymous Variables in Rules

Suppose that we are interested in whether paul is a murderer, but we are not interested in who he has murdered.

We have a database of facts, including:

**murdered(john, paul).**

**murdered(paul, john).**

**murdered(paul, beer).**

We could type the query: **?- murdered(paul,X)** to see if paul is a murderer but it is better to define a new rule with an anonymous variable:

**murderer(X) :- murdered(X,\_)**

# Multiple Anonymous Variables

We can have >1 anonymous variable in a clause. For example:

**? - parent(,\_)**

This is asking if there are any parent relationships in the database

**Note:** each of the “\_” means a different logical variable. So, this query is equivalent to **parent(X,Y)** and not **parent(X,X)**

# Arithmetic Computation in Prolog

- Prolog: designed for *symbolic* rather than *numeric* computation
- Not good for numeric problems
- Imperative languages are good at numeric but terrible at symbolic
- Functional languages are good at both.
- Prolog provides a bare minimum:
  - +
  - -
  - \*
  - /
  - div --- *Integer* division
  - mod --- *Integer* remainder

# Unexpected Behaviour

Operators do not behave exactly as expected:

?- 3 +4.

**no**

?-

So far as Prolog is concerned, “3+4” is an expression corresponding to a *structure* (see later). A structure cannot be proved from a database and so the response is ‘**no**’.

We must tell Prolog to treat the structure as an arithmetic expression and actually evaluate it. We could try:

?- X = 3+4.

**X = 3+4?**

**yes**

The ‘=’ operator simply means ‘*do these 2 terms match*’. Prolog says they match if X is the term “3+4” ... (not very useful!)

# Comparison and assignment

Do not confuse Prolog's '=' with Java's '='. They are very different. Prolog's is the matching operator (a bit like '==' in JAVA ... see later)

We must use the 'assignment' operator 'is':

**?- X is 3 +4.**

**X = 7?**

**yes**

The 'is' is only superficially like '=' in JAVA.

There must be a logical variable on the left hand side and an arithmetic expression on the right hand side. The 'is' tells Prolog to evaluate the RHS and match the result with the variable on the left.

We can have more complex expressions:

**?- X is 1+2\*3/4-5**

**X = -2.5?;**

**no**

# More Complex Expressions

We can have expressions containing other logical variables:

**?- X is 3 +4, Y is X+X.**

**Y = 14**

**X =7 ?**

**yes**

Note: the order of the sub-goals is important:

**?- Y is X+X, X is 3+4.**

**{INSTANTIATION ERROR: \_36 is \_34+\_34 - arg2}**

Here, sub goals are evaluated from left to right and because X has not yet been instantiated, the query cannot be evaluated

‘34’ and ‘36’ are the internal names that Prolog gives to variables X and Y !!!

# What is is?

For 'is' to work, all the logical variables in the expression on the right hand side must be instantiated.

We can't use 'is' backwards --- 'is' is functional not relational

This only works one way: for efficiency purposes.

Conceptually, there is no reason why 'is' shouldn't be able to work backwards: but it would be very inefficient ...

Question: can you see why?

Example (not real Prolog) of what would be nice:

**?- 64 is X \*Y**

**Y=64**

**X=1?;**

**Y = 32**

**X = 2?; ...**



# Other arithmetic operators

The usual arithmetic comparison operators are also available:

`==` Equal

`!=` Not equal

`<`, `>`, `>=`, `=<` greater/less...

As with 'is', these will not work if there are uninstantiated logical variables in the expressions being compared

Usually Prolog programmers do not use '`==`' and '`!=`'. Instead, the general matching and non-matching operators '`=`' and '`\=`' are used.

# Is is not assignment

Despite the superficial similarity, the ‘is’ operator is not an assignment:

```
?- 64 is 32 * 2. %1
```

**yes**

```
?- X is 64, X is 32 * 2. %2
```

```
X = 64?
```

**yes**

```
?- X is 64, X is 1+2. %3
```

**no**

%1 -- 64 matches the result of 32\*2

%2 -- The first subgoal succeeds, instantiating X to 64. The expression in the second subgoal evaluates to 64 and since the values on either side of the ‘is’ can be matched, yes is returned

%3 -- 1st subgoal succeeds, in second subgoal the expressions cannot be matched and so Prolog returns no.

# Is is still not assignment

?- X is X+1.                    %4

{INSTANTIATION ERROR: \_36 is \_33+1 -arg 2}

?- X is 64, X is X +1.            %5

no

%4 -- The system cannot evaluate the RHS 'X+1' because X has not been instantiated, and so the query fails

%5 -- The 1st subgoal succeeds and instantiates X to 64. X+1, on the RHS of the 2nd subgoal is evaluated to 65. 64 and 65 cannot be matched across the 'is' and so the result is no

## *is* example: the size of a list

```
size([],0).  
size([H|T],N) :- size(T,N1), N is N1+1.  
% or size([_|T],N) :- size(T,N1), N is N1+1.  
  
| ?- size([1,2,3,4],N).
```

N = 4

# Some complex queries and data structures

```
course(pl2, se109, gibson, paul, 'pgibson@cs.may.ie',  
salesian, 117, Tuesday, 10, callan, slt).
```

To find out when I lecture I write the query:

```
?- course(_,_,gibson,paul,_,_,_,Day,Hour,_,_).
```

```
Hour = 11
```

```
Day = tuesday
```

To find out what and where I am lecturing I could use the following query:

```
?- course(Name,_,gibson,paul,_,_,_,_,Building,Room).
```

```
Room = slt
```

```
Building = callan
```

```
Name = pl2
```

# Without structures

- This is very clumsy
- The course has 12 arguments
- Must remember the order of arguments (no type checking to help)
- Which room is lecturer's office and which is lecture theatre
- Things would be better if we could group related data. For example:

**course(Description, Lecturer, Time, Room).**

requires only 4 arguments and can be done using Prolog structures

- For example:
  - the data about the course description could be written as **description(pl2, se109).**
  - the data for a Room as **room(callan, slt).**
  - the data for Time as **time(Tuesday 10).**

# The new structured data base entry

```
course( description(pl2,se109),  
        lecturer(gibson, paul, 'pgibson@cs.may.ie', room(salesian, 117)),  
        time(tuesday,10),  
        room(callan,slt) ).
```

**Note:** the use of the same room structure for offices and lecture theatres

**Advantages:** program easier to understand, queries easier to write, eg:

```
?- course(_lecturer(gibson,paul,_,_), Time, _).
```

```
Time = time (Tuesday,10)
```

**Disadvantages:** it is longer

# Structured Queries

We can now use the course structure to define a new relation. For example, to test if a room is occupied at a given time:

**occupied(Room,Time) :- course(\_,\_,Time, Room).**

Now we can write a query:

**?- occupied(room(callan,slt), time(tuesday,10)).**

**yes**

Or another to find out when a room is occupied:

**?- occupied(room(callan,slt),Time).**

**Time = time(tuesday,10)?**

Or another to find what room is occupied at a given time:

**?- occupied(Room, time(tuesday,10)).**

**Room = room(callan,slt)?**



# Structure classes and types

The ‘type’ of a structured object is defined by:

- its functor name
- its arity -- number of components

Thus,  $f(a,b,c)$  and  $f(a,b,c,d)$  are objects of the same ‘class’ but of different ‘type’ within the ‘class’.

Now, we can have many different types in the same class:

**student(fred,123,taught,pl2,55)**

**student(paul,research)**

Note: the absence of type checking could lead to problems. For example, having a research student on a taught course:

**student(paul,123,research,pl2,65)**

# Processing such structures

Assuming that we don't make typing errors, processing these 'variant records' (the term is common in imperative languages) is simple:

```
pass(student(Name,_,_,_,Mark)):- Mark>=60.
```

Now we can test if paul and fred have passed their exams:

```
?- pass(student(fred,123,taught,pl2,65)).
```

**yes**

```
?- pass(student(paul,research)).
```

**no**

This last query fails because the structure has a different number of arguments to that to which we are trying to match.

Note: if we make a mistake the system may not notice:

```
?- pass(student(drinks,lots,of,beer,80)).
```

**yes**

Functional languages give all the flexibility of structures plus safe type checking

# Simple Input/Output

## **Write** predicate

`write( )` Writes a single term to the terminal.

`write_ln()` Writes a term to the terminal followed by a new line.

`tab(X)` Writes an `X` number of spaces to the terminal.

# Simple Input/Output

## Read predicate

- **read(X)** Reads a term from the keyboard and instantiates variable X to the value of the read term.
- This term to be read has to be followed by a dot “.” and a white space character (such as an enter or space).
- For example:

hello :-

```
write('What is your name ?'),  
read(X),  
write('Hello'), tab(1), write(X).
```

# Assertions

- **assert(X)** Adds a new fact or clause to the database. Term is asserted as the last fact or clause with the same key predicate.
- **asserta(X)** Same as assert, but adds a clause at the beginning of the database
- **assertz(X)** Exactly the same as **assert(X)**

For example:

```
:- dynamic good/2.
```

```
:- dynamic bad/2.
```

```
assert(good(skywalker, luke)).
```

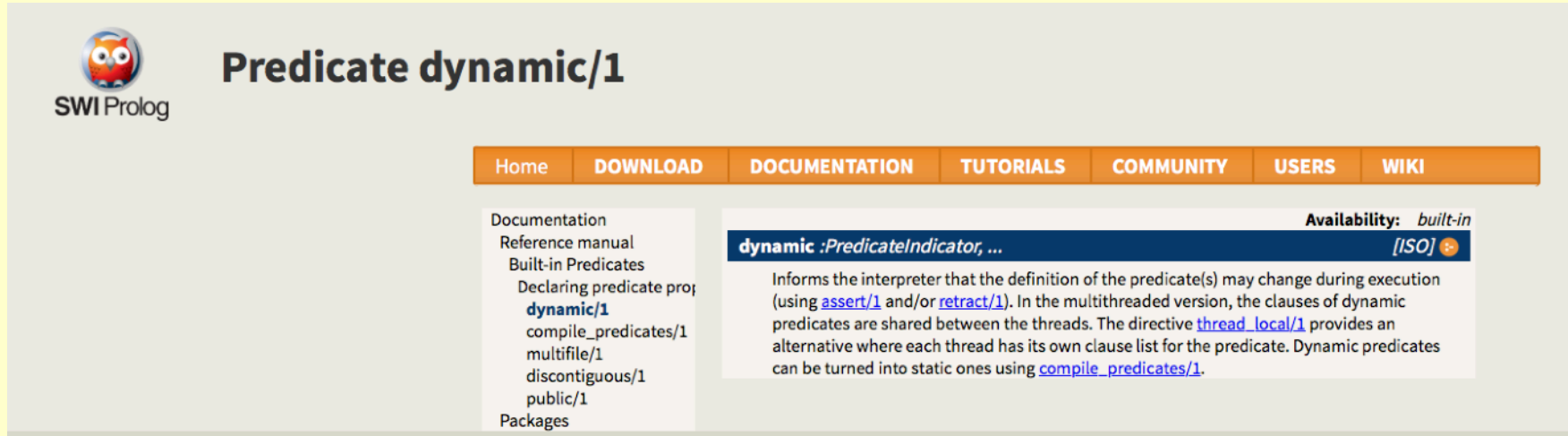
```
assert(good(solo, han)).
```

```
assert(bad(vader, darth)).
```

Question: why do you think we need to use “dynamic”?

# Assertions

Question: why do you think we need to use “dynamic”?



The screenshot shows the SWI Prolog documentation page for the `dynamic/1` predicate. The page features a navigation bar with links for Home, DOWNLOAD, DOCUMENTATION, TUTORIALS, COMMUNITY, USERS, and WIKI. A sidebar on the left lists various documentation topics, including 'dynamic/1'. The main content area displays the `dynamic` predicate signature as `dynamic :PredicateIndicator, ...` with an availability of `built-in` and an ISO standard icon. The text explains that this predicate informs the interpreter that the definition of the predicate(s) may change during execution, and it is used in conjunction with `assert/1` and `retract/1`.

SWI Prolog

## Predicate dynamic/1

Home DOWNLOAD DOCUMENTATION TUTORIALS COMMUNITY USERS WIKI

Documentation  
Reference manual  
Built-in Predicates  
Declaring predicate pro:  
**dynamic/1**  
compile\_predicates/1  
multifile/1  
discontiguous/1  
public/1  
Packages

**Availability:** *built-in*  
**dynamic** :PredicateIndicator, ... [ISO]

Informs the interpreter that the definition of the predicate(s) may change during execution (using [assert/1](#) and/or [retract/1](#)). In the multithreaded version, the clauses of dynamic predicates are shared between the threads. The directive [thread\\_local/1](#) provides an alternative where each thread has its own clause list for the predicate. Dynamic predicates can be turned into static ones using [compile\\_predicates/1](#).

<http://www.swi-prolog.org/pldoc/man?predicate=dynamic/1>

# Retractions

- **retract(X)** Removes fact or clause X from the database.
- **retractall(X)** Removes all facts or clauses from the database for which the head unifies with X.
- For example:  
retract(bad(vader, darth)).  
retractall(good(\_, \_)).  
?- good(X, Y).  
No

# Cuts

What does the following Program program do:

```
max(X,Y,Y):- X =< Y.
```

```
max(X,Y,X):- X>Y.
```

```
?- max(2,3,Max).
```

```
Max = 3
```

```
yes
```

```
?- max(2,1,Max).
```

```
Max = 2
```

```
yes
```

Why is it considered to be inefficient?



# Cuts

What happens if at some stage backtracking is forced?

The program will try to re-satisfy  $\max(3,4,Y)$  using the second clause. This is completely pointless: the maximum of 3 and 4 is always 4. There is no second solution to find.

In other words, the two clauses  $(X \leq Y)$  and  $(X > Y)$  in the above program are mutually exclusive: if the first succeeds, the second must fail and vice versa.

So attempting to re-satisfy this clause is a complete waste of time. We use a cut to stop this from happening

# Cuts - using ,!.

Here is a more efficient version:

```
max(X,Y,Y) :- X =< Y,!.  
max(X,Y,X) :- X>Y.
```

Notice the cut

```
?- max(2,3,Max).
```

```
Max = 3  
yes
```

```
?- max(2,1,Max).
```

```
Max = 2  
yes
```

Note that this cut does not change the meaning of the program. Our new code gives exactly the same answers as the old one, but it's more efficient. In fact, the program is exactly the same as the previous version, except for the cut, and this is a pretty good sign that the cut is a sensible one. Cuts like this, which don't change the meaning of a program, have a special name: they're called **green** cuts.

# Negation and cuts

## Negation as Failure

Negation in Prolog is implemented based on the use of cut. Actually, negation in Prolog is the so-called *negation as failure*, which means that to negate  $p$  one tries to prove  $p$  (just executing it), and if  $p$  is proved, then its negation,  $\text{not}(p)$ , fails. Conversely, if  $p$  fails during execution, then  $\text{not}(p)$  will succeed. The implementation of  $\text{not}/1$  is as follows:

```
not(Goal) :- call(Goal), !, fail.  
not(Goal).
```

(`fail/0` is a builtin predicate which always fails. It can be trivially defined as `fail:- a = b.`)

# Negation and cuts: difficult for beginners

```
unmarried_student(X):-  
    not(married(X)), student(X).  
  
student(joe).  
married(john).
```

This program seems to suggest that `joe` is an unmarried student, and that `joe` is not an unmarried student, and indeed:

```
?- unmarried_student(joe).  
yes  
?- unmarried_student(john).  
no
```

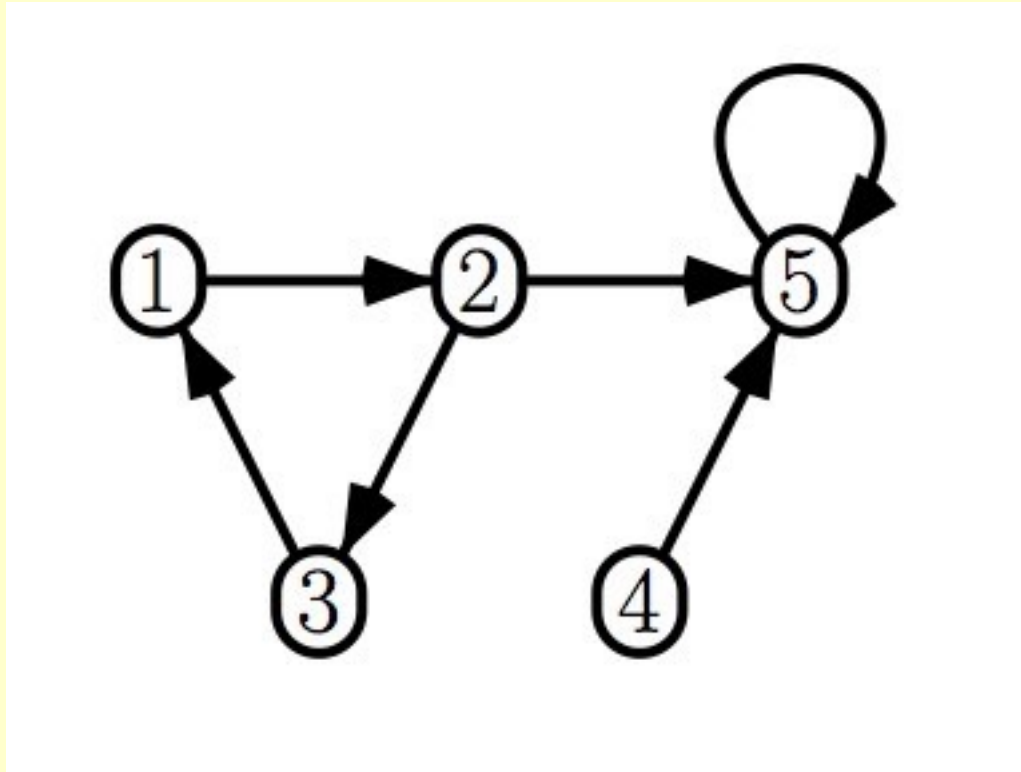
But, for logical consistence, asking for unmarried students should return `joe` as answer, and this is not what happens:

```
?- unmarried_student(X).  
no
```

The reason for this is that the call to `not(married(X))` is not returning the students which are not married: it is just failing because there is at least a married student.

# Prolog Programming Problem 1 (relatively easy)

Check if there is a path between two nodes in a directed graph



# Prolog Programming Problem 2 (more difficult)

## Arithmetic Expression Search

Given a list of positive integers, and a target

Find an integer expression using operators  $+$ ,  $-$ ,  $*$

such that a subset of the numbers can be combined (using each number at most 1 time) to reach the target

For example, list 1,3,10,40 target 29:  $(3*10)-1$   
target 5: false