Prolog 1

MSc Computing

Fariba Sadri

With thanks to Keith Clark for the use of some of his lecture material

Prolog

Prolog is a high level *declarative* programming language based on a subset of predicate logic. It is a *logic programming* language.

Particularly favoured for applications in
- AI
- expert system and
- computational linguistics.

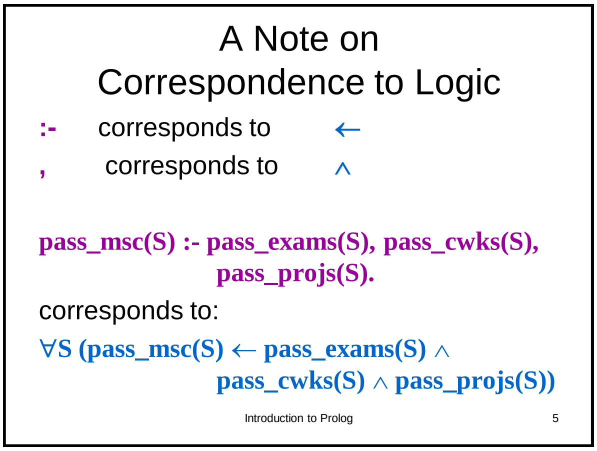
It was developed in the early 1970s through
- the theoretical studies of Professor Robert Kowalski at Imperial College and Edinburgh
- Alain Colmerauer in Marseille, France, and
- the first compiler was written by David H.D. Warren in Edinburgh, Scotland.

- We will be using Sicstus Prolog and Windows. You can use Linux.
- Program files are saved as plain text.
- Prolog tutorials in lab in weeks 5-7. Please see the timetable.
- Assessment is by assessed lab exercises + Lab examination in Jan
- Possible Mock test in week 11 (unassessed)

Example: A very short Prolog program

```
% A set of facts:
pass_exams(john).
pass_cwks(john).
pass_projs(john).

% A rule:
pass_msc(S) :- pass_exams(S), pass_cwks(S), pass_projs(S).
```



A Note on Correspondence to Logic

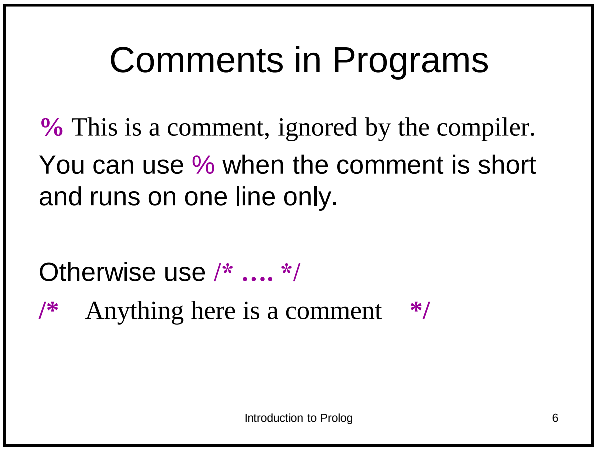
:- corresponds to ←

, corresponds to ∧

pass_msc(S) :- pass_exams(S), pass_cwks(S),
pass_projs(S).

corresponds to:

∀S (pass_msc(S) ← pass_exams(S) ∧
pass_cwks(S) ∧ pass_projs(S))



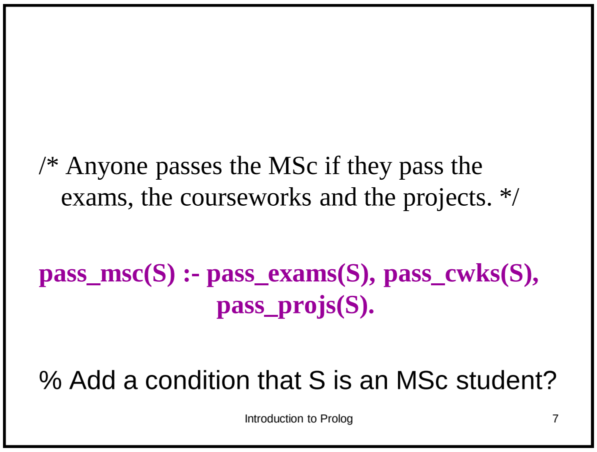
Comments in Programs

% This is a comment, ignored by the compiler.

You can use % when the comment is short and runs on one line only.

Otherwise use /* **….** */

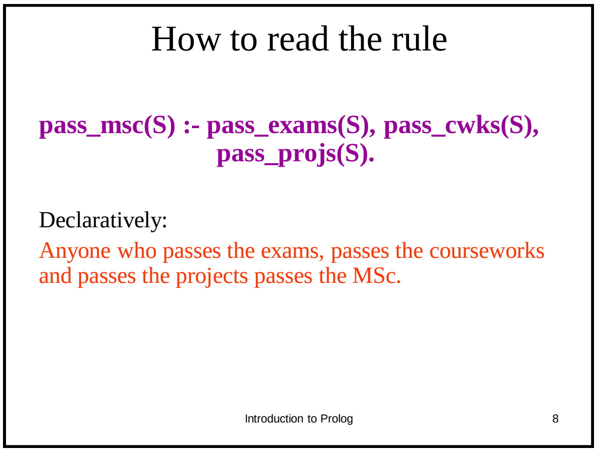
/* Anything here is a comment */



/* Anyone passes the MSc if they pass the exams, the courseworks and the projects. */

pass_msc(S) :- pass_exams(S), pass_cwks(S),
pass_projs(S).

% Add a condition that S is an MSc student?



How to read the rule

pass_msc(S) :- pass_exams(S), pass_cwks(S),
pass_projs(S).

Declaratively:

Anyone who passes the exams, passes the courseworks and passes the projects passes the MSc.

Procedurally:
There are two readings:

1. To show that someone passes the MSc show that he/she passes the exams, passes the courseworks and passes the projects.
2. To find who passes the MSc find who passes the exams, the courseworks and the projects.

Example Queries to the Program

```
pass_exams(john).
pass_cwks(john).
pass_projs(john).
pass_msc(S) :- pass_exams(S), pass_cwks(S), pass_projs(S).
```

```
Query:    pass_msc(john)?
Answer:   yes
Query:    pass_msc(mary)?
Answer:   no
Query:    pass_msc(X)?    (who passes the MSc?)
Answer:   X = john
```

Prolog syntax

A **Prolog program** is a sequence of ***clauses***.

A clause has the form:

	H :- C_1, ..., C_k.	*conditional clause*
or	**H.**	*unconditional clause*

A terminating
'.<space>',
'.<newline>' or
'.<tab>'
is *essential* after each clause.

Prolog syntax cntd. H :- C_1,...,C_k.

H and each **C_i** is an *atomic formula* of the form:

p(t_1,..., t_n) or **p**

Must be NO space between p and the (

p is the predicate or relation name of the atomic formula. **t_1,..., t_n** are *terms*.

Clause is *about* the predicate of **H.**

Each **C_i** is sometimes referred to as a *call* or *condition*.

Later we will see that we can have more complex conditions.

Logical reading

A conditional clause

H :- C1,...,Ck. is read as:

$\forall X_1 \ldots X_m (H \leftarrow C_1 \wedge \ldots \wedge C_k)$

where the X_i are *all* the variables that occur in the clause,

or equivalently:

$\forall X_1 \ldots X_i (\exists X_{i+1},\ldots,\exists X_m (H \leftarrow C_1 \wedge \ldots \wedge C_k))$

where $X_{i+1},\ldots,X_m$ are variables that only appear in the conditions of the clause.

An unconditional clause

H is read as:

$\forall X_1 \ldots X_m (H)$

where the X_i are *all* the variables that occur in **H.**

Prolog terms

- *Constants* - usually alphanumeric sequence of one or more symbols beginning with a *lower case letter*, and possibly containing _

  e.g. **bill**, **maryJones**, **mary_jones, elephant67**

- *Numbers* - usual syntax e.g. **3**, **-6**, **34.89**

- *Variable names* - alphanumeric sequence of one or more symbols beginning with an upper case letter or _ e.g. **X**, **Apple**, **_456**, **_**

- *Compound terms* - a *function* name (same syntax as constant) applied to n terms of the form **f(t1,..,tn)**,

  e.g.

  name(First_name, Surname)
  dep_rep (Department, Degree, Year)
  dep_rep (computing, msc, 2014)

appointed(name(alex, polise),
dep_rep(computing, msc, 2014))
appointed(name(sean, rankine),
dep_rep (computing, msc, 2014))

Alternatively:

appointed(alex, polise, dep_rep, computing, msc, 2014)

appointed_dep_rep(alex, polise, computing, msc, 2014)

Predicate names have same syntax as constants, i.e.

alphanumeric sequence of one or more symbols beginning with a *lower case letter*, and possibly containing _

E.g. pass_msc

appointed

rep2014

More on syntax

Constants, function symbols and predicate symbols can also be *any* sequence of characters in single quotes, e.g.

’fs@doc.ic.ac.uk ’

’Sam ’

’bill green’

’***’**

There are two other kinds of terms,

strings and

lists

(we will come to these later).

Facts and Rules

If an unconditional clause:

H.

contains *no* variables then the clause is called a **fact.**

E.g. **pass_cwks(john).**

no_of_children(mary, 3).

All other Prolog clauses are called **rules.**

E.g.

drinks(john) :- anxious(john).

anxious(X):- has_driving_test(X).

covers(sky, X).

Prolog queries

A **query** is a conjunction of conditions, i.e.

?- C_1, . . . , C_n .<newline>

Each **C_i** is a condition/call (as in a clause).

?- is a prompt displayed by Prolog.
Terminating **.**<newline> is needed.

Prolog queries cntd

?- C_1, . . . , C_n .<newline>

If it contains variables, the query is a request for a substitution (a set of term values) θ for the variables of the query such each of the conditions:

C_1θ, . . . , C_nθ

is a logical consequence of the program clauses, or for a confirmation that there is no such θ.

C_i θ is **C_i** with any variable in **C_i** (given a value in θ**)** replaced by its assigned value.

If there are no vars in query, then the query is a request for a report on whether or not the query, as given, is a logical consequence of the program clauses.

Exercise

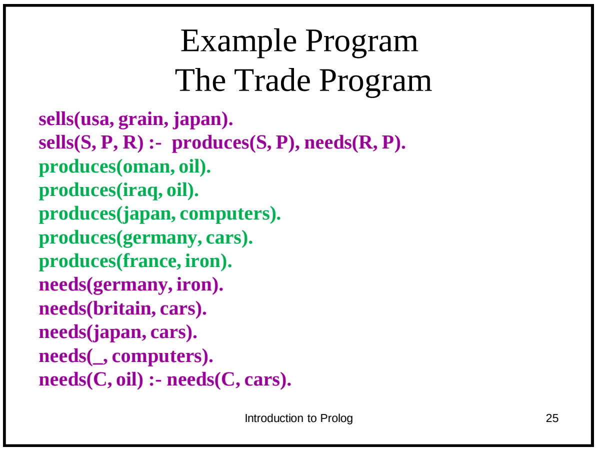
C	θ	C θ
p(X)	{X=john}	p(john)
q(X,Y)	{X=1, Y=2}	
q(X,Y)	{X=1, Y=f(Z)}	
q(X, f(X))	{X=g(5)}	

Example query

?- pass_msc(X)

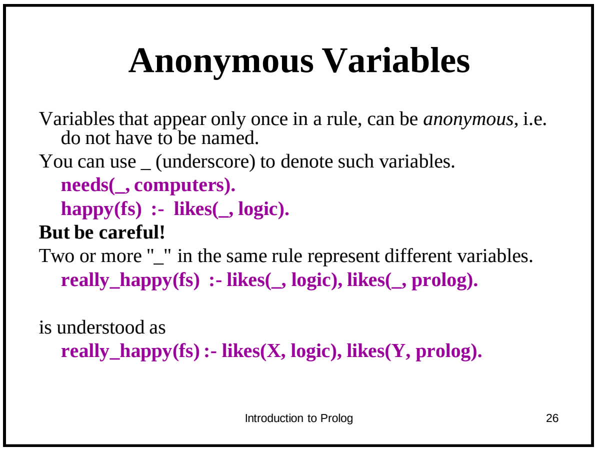
i.e. "Is there someone, X, who passes the MSc?"
or "Who passes the MSc?
It is a request for an answer
θ **={X=*name*}**
such that
pass_msc(X)θ
i.e. **pass_msc(*name*)**

follows from the program clauses *or*
for confirmation that there is no such θ (no such name).



Example Program
The Trade Program

```
sells(usa, grain, japan).
sells(S, P, R) :-  produces(S, P), needs(R, P).
produces(oman, oil).
produces(iraq, oil).
produces(japan, computers).
produces(germany, cars).
produces(france, iron).
needs(germany, iron).
needs(britain, cars).
needs(japan, cars).
needs(_, computers).
needs(C, oil) :- needs(C, cars).
```



Anonymous Variables

Variables that appear only once in a rule, can be *anonymous*, i.e. do not have to be named.

You can use _ (underscore) to denote such variables.

```
needs(_, computers).
happy(fs)  :-  likes(_, logic).
```

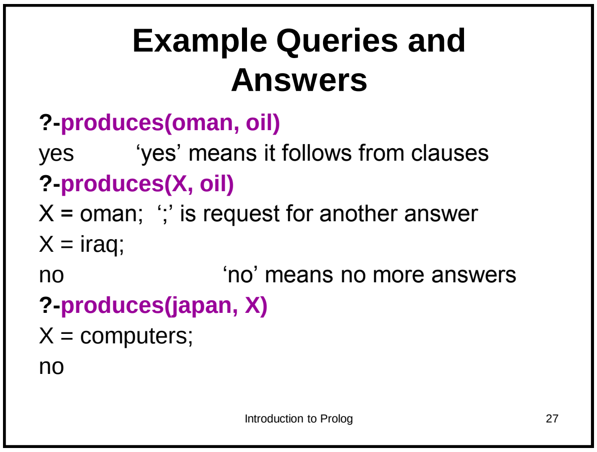
But be careful!

Two or more "_" in the same rule represent different variables.

```
really_happy(fs)  :- likes(_, logic), likes(_, prolog).
```

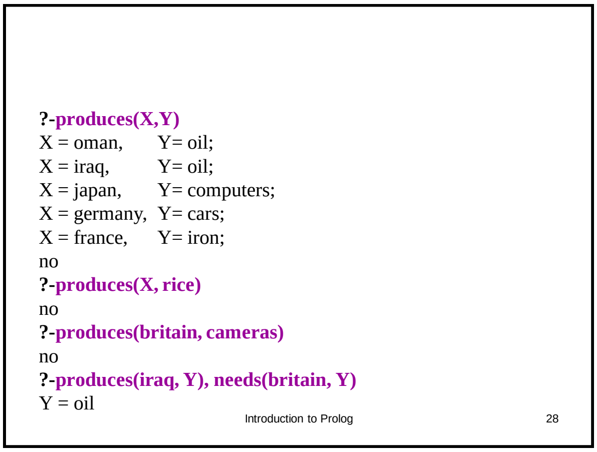
is understood as

```
really_happy(fs) :- likes(X, logic), likes(Y, prolog).
```



Example Queries and Answers

```
?-produces(oman, oil)
yes          'yes' means it follows from clauses
?-produces(X, oil)
X = oman;  ';' is request for another answer
X = iraq;
no                    'no' means no more answers
?-produces(japan, X)
X = computers;
no
```



```
?-produces(X,Y)
X = oman,     Y= oil;
X = iraq,     Y= oil;
X = japan,    Y= computers;
X = germany,  Y= cars;
X = france,   Y= iron;
no
?-produces(X, rice)
no
?-produces(britain, cameras)
no
?-produces(iraq, Y), needs(britain, Y)
Y = oil
```

Exercise: Trade Program

Write Prolog Queries for the following:

1. **Does Britain sell oil to the USA?**
2. **Who sells grain to who?**
3. **Who sells oil to Britain?**
4. **Who sells what to Germany?**
5. **Who sells something to Germany?**

Exercise Trade Program ctnd.

6. **Which two countries have mutual trade with one another?**

7. **Which two different countries have mutual trade with one another? (X\=Z means X and Z are different from one another.)**

8. **Express a prolog rule for "bilateral_traders(X,Z)" such that X and Z are two different countries that have mutual trade with one another.**

9. **Express the following query in Prolog.**

Who produces something that is needed by both Britain and Japan?

What answer(s) will Prolog give?

Scope of identifiers

- The scope of a variable is just the clause or query in which it occurs.
- The scope of any other name (constant, function name, predicate name) is the whole program and any query.

Example Program Work-Manager

```
% worksIn(Person, Department)
    worksIn(bill, sales).
    worksIn(sally, accounts).

% deptManager(Department, Manager)
    deptManager(sales, joan).
    deptManager(accounts, henry).

% managerOf(Worker, Manager)
    managerOf(joan, james).
    managerOf(henry, james).
    managerOf(james, paul).
```

Exercise

1. Define colleague/2, such that *colleague(W1,W2)* holds if W1,W2 are *different* workers that work in the same department.
2. Add a new clause for *managerOf(W,M)* to express that M is the manager of W if M is the manager of the department in which W works.

Recursion

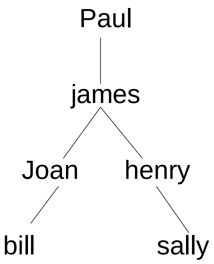
```
superiorOf(E,S) :-    managerOf(E,S).
superiorOf(E,S):-     managerOf(E,M),
                      superiorOf(M,S).
```

superiorOf/2 is a ***recursive*** predicate.
The first rule for **superiorOf**/2 is a ***base case***.
The second rule for **superiorOf**/2 is a ***recursive rule***.

With earlier facts and rules we get:

```
?-superiorOf(bill, paul).
Yes
```

What are the answers to ?-superiorOf(X,Y). ?



Paul

james

Joan

henry

bill

sally

Disjunction in bodies of rules and queries

In Prolog **;** is the same as the logical symbol $\vee$.
E.g.

inelligible_to_vote(X) :-under_age(X) ; in_prison(X).

The Prolog rule

p:-c1;c2.

has the same meaning as the two rules

p:-c1.
p:-c2.

Exercise: Prove in logic that

$$p \leftarrow c1 \vee c2 \equiv (p \leftarrow c1) \wedge (p \leftarrow c2).$$

So

inelligible_to_vote(X) :- under_age(X) ;
in_prison(X).

Can be written as:

inelligible_to_vote(X) :- under_age(X).
inelligible_to_vote(X) :- in_prison(X).

Arithmetic

- is/2 is a primitive Prolog predicate for evaluating arithmetic expressions.
- The call
  X is Exp
where Exp is an arithmetic expression, *unifies* X with the value of Exp
- Operators work in the same way as in most languages + - * /
- X can be a number or an unbound variable but not another expression.
- Note that at the time of evaluation of condition
  X is Exp, Exp must be *ground*, i.e. contain no unbound vars.
- Arithmetic values can be compared using built in relations:
  <, =< , >, >=

Arithmatic Examples

- X is 2*4 (unifies/binds X to 8)
- W=4,..., U is 25*W, ..., X is U/5
  (unifies/binds U to 100, and X to 20)
- X is 4, X is X+1 (will fail!)
- X is 4, NewX is X+1
  (unifies/binds NewX to 5)
- The difference between is and =.
  Try X is 2+1, Y=2+1.

Example: Factorial

The Factorial of a non-negative, non-zero integer N, denoted N!,

is the product of N and all the non-negative, non-zero integers below it.

Factorial(N) =1 if N=0
Factorial(N) = N*Factorial(N-1) if N>0

Factorial in Prolog

Factorial(N) =1   if N=0
Factorial(N) = N*Factorial(N-1)   if N>0

In Prolog:

```
fact(0,1).
        \* we can also write this as:
        fact(N, FN):- N=0, FN=1.   */
fact(N, FN):- N>0, X is N-1, fact(X,FX),
                        FN is N*FX.
```

Example Uses

Find the factorial of a number

```
        ?- fact(4,X).
        X=24
```

Check the factorial of a number

```
        ?- fact(3,6)
        yes
```

Combined in any conjunction

```
        ?- fact(4, X), fact(6, Y), Y is 30*X.
        X = 24, Y = 720      yes
```

Cannot use invertibly:

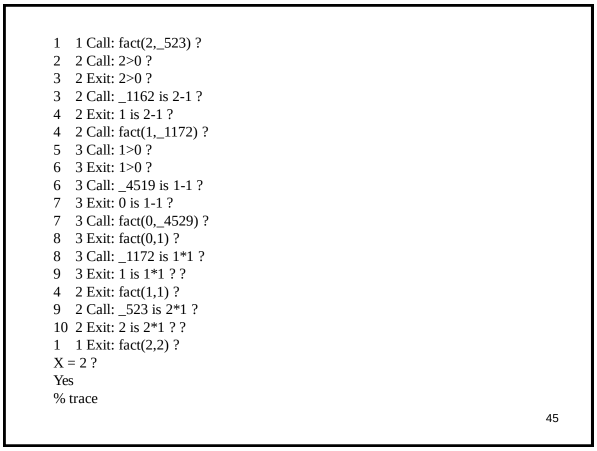
```
?- fact(X,2).
! Instantiation error in argument 1 of >/2
```

because the condition: **N > 0** needs **N** to be known.

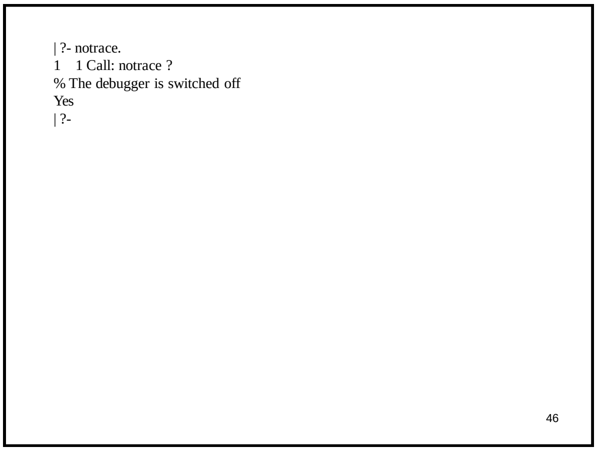
trace / notrace

```
| ?- fact(3,X).
X = 6 ?
Yes

| ?- trace.
% The debugger will first creep -- showing everything
   (trace)yes% trace
| ?- fact(2,X).
```



```
1    1 Call: fact(2,_523) ?
2    2 Call: 2>0 ?
3    2 Exit: 2>0 ?
3    2 Call: _1162 is 2-1 ?
4    2 Exit: 1 is 2-1 ?
4    2 Call: fact(1,_1172) ?
5    3 Call: 1>0 ?
6    3 Exit: 1>0 ?
6    3 Call: _4519 is 1-1 ?
7    3 Exit: 0 is 1-1 ?
7    3 Call: fact(0,_4529) ?
8    3 Exit: fact(0,1) ?
8    3 Call: _1172 is 1*1 ?
9    3 Exit: 1 is 1*1 ? ?
4    2 Exit: fact(1,1) ?
9    2 Call: _523 is 2*1 ?
10  2 Exit: 2 is 2*1 ? ?
1    1 Exit: fact(2,2) ?
X = 2 ?
Yes
% trace
```



```
| ?- notrace.
1    1 Call: notrace ?
% The debugger is switched off
Yes
| ?-
```