### Imperative vs Declarative Programming

#### CSC9Y4

**Imperative Programming**
- **Paradigm**: How to **SOLVE** the problem
- **Main operation**: Assignment (transform a state)
- **Program**: A sequence of instructions
- **Execution env**: Compiled to a low-level machine
- **Languages**: Fortran C/C++, Pascal Ada, Java*
- **Features**: Optimised, efficient code

**Declarative Programming**
- **Paradigm**: How to **DEFINE** the problem
- **Main operation**: Function application, Deductive step (stateless*)
- **Program**: A set of definitions (a knowledge base)
- **Execution env**: Interpreted* (function applications, deduction…)
- **Languages**: Lisp, ML, Haskell, Prolog*
- **Features**: Expressive, provably correct

---

```java
private int[] array; ...
private void quicksort(int low, int high) {
    int i = low, j = high;
    int pivot = array[low + (high-low)/2];
    while (i <= j) {
        while (array[i] < pivot) i++;
        while (array[j] > pivot) j--;
        if (i <= j) {
            exchange(i, j);
            i++; j--;
        }
    }
    if (low < j) quicksort(low, j);
    if (i < high) quicksort(i, high);
}
```

```prolog
quicksort([X|Xs], Ys) :-
    partition(Xs, X, Littles, Bigs),
    quicksort(Littles, Ls),
    quicksort(Bigs, Bs),
    append(Ls, [X|Bs], Ys).
quicksort([], []).
```

```prolog
partition([X|Xs], Y, [X|Ls], Bs) :-
    X <= Y, partition(Xs, Y, Ls, Bs).
partition([X|Xs], Y, La, [X|Bs]) :-
    X > Y, partition(Xs, Y, La, Bs).
partition([], _, [], []).
```

---

```java
private int[] array;
...
```

```java
private void quicksort(int low, int high) {
    int i = low, j = high;
    int pivot = array[low + (high-low)/2];
    while (i <= j) {
        while (array[i] < pivot) i++;
        while (array[j] > pivot) j--;
        if (i <= j) {
            exchange(i, j);
            i++; j--;
        }
    }
    if (low < j) quicksort(low, j);
    if (i < high) quicksort(i, high);
}
```
What is logic?

The branch of philosophy concerned with analysing the patterns of reasoning by which a conclusion is drawn from a set of premises, without reference to meaning or context 
(Collins English Dictionary)

Logic is concerned with two key skills, which any computer engineer or scientist should have:
– Abstraction
– Formalisation

Logic is a formalisation of reasoning. Formal language for deducing knowledge from a small number of explicitly stated premises (or hypotheses, axioms, facts). Formal framework for representing knowledge. Differentiates between the structure and content of an argument

Knowledge can be stated concisely and precisely. The process of reasoning from that knowledge can be made rigorous.

What is an argument?

An argument is just a sequence of statements. Some of these statements, the premises, are assumed to be true and serve as a basis for accepting another statement of the argument, called the conclusion.

If the conclusion is justified, based solely on the premises, the process of reasoning is called deduction, e.g.

“Alexandria is a port or a holiday resort.
Alexandria is not a port.
Therefore, Alexandria is a holiday resort”

If the validity of the conclusion is based on generalisation from the premises, based on strong but inconclusive evidence, the process is called induction, e.g.

“Most students who have blue eyes also have blonde hair.
John has blue eyes.
Therefore John has blonde hair.”

Propositional logic (Boole, 1815-1864)

Simple types of statements, called propositions, treated as atomic building blocks for more complex statements, e.g.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alexandria is a port or a holiday resort.</td>
<td>P or H</td>
</tr>
<tr>
<td>Alexandria is not a port.</td>
<td>Not P</td>
</tr>
<tr>
<td>Therefore, Alexandria is a holiday resort.</td>
<td>(P or H) ∧ ¬P → H</td>
</tr>
<tr>
<td>It is hot.</td>
<td>R</td>
</tr>
<tr>
<td>It is humid.</td>
<td>S</td>
</tr>
<tr>
<td>It is raining.</td>
<td>T</td>
</tr>
<tr>
<td>If it is humid, then it is hot.</td>
<td>S → R</td>
</tr>
<tr>
<td>If it is hot and humid, then it is raining.</td>
<td>(R ∧ S) → T</td>
</tr>
</tbody>
</table>
Logical constants: true, false

Propositional symbols: P, Q, S, T ... (atomic sentences)

Wrapping parentheses: ( ... )

Sentences are combined by connectives:

- $\land$ and
- $\lor$ or
- $\neg$ not
- $\Rightarrow$ implies
- $\Leftrightarrow$ is equivalent

$\neg S$, (S $\lor$ T), (S $\land$ T), (S $\rightarrow$ T), and (S $\leftrightarrow$ T) are sentences

Propositional logic (Boole, 1815-1864)

A simple language useful for showing key ideas and definitions, where the user defines the semantics of each propositional symbol, e.g.

P means “It is hot”

Hard to identify “individuals”, relationships and generalizations, like

John is a man and John is a builder.
All triangles have 3 sides.

Predicate logic (Frege 1848-1925)

First-Order Logic (FOL): extension of propositional logic.

A predicate is just a property. Predicates define relationships between any number of entities using quantifiers and variables, e.g.

Every elephant is grey $\forall X (\text{elephant}(X) \rightarrow \text{grey}(X))$

There is a white alligator $\exists X (\text{alligator}(X) \land \text{white}(X))$

With $\forall$ “for all”, “for every”, and $\exists$ “there exists”, X variable and elephant predicate

First-Order Logic (FOL) is expressive enough to concisely represent this kind of information, but “harder” computationally.

PROgramming in LOGic - PROLOG

CSC9Y4
PROgramming in LOGic

Prolog is a language which embodies the logic programming paradigm. Logic as the basis for a programming language is a great idea since it’s aimed at the problem, not the machine. Used for an extensive set of applications.

How is the logic programming paradigm different from the imperative programming paradigm?

– What is a program?
– Are there variables?
– How are they manipulated?
– Are there modules?
– Is there branching and iteration?
– What data structures are there?

The Prolog Interpreter

Prolog is a language with an interpreter. The user interacts with the Prolog Interpreter, which relies on a knowledge base, e.g.

Yes.           Yes.           No.

?- X is 4 * (5 + 2).       ?- length([3,7,1,4],L).

X and L are variables, but values of variables do not persist after the interpreter has reported them.

?- X is 3 + 2, Y is 5 * 4, Z is X + Y.
X and Y carry values from the first (left to right) two (comma separated) subgoals into the third subgoal. Current values of X, Y and Z will not persist to the next query!

The Prolog Database

During an interactive session, the Prolog interpreter has a temporary store, called the Prolog database (just 'the database' for short). In this store it keeps facts and rules.

When you start a session, the database is empty, and there are certain ways in which it can be added to.

The contents of the database are the 'knowledge' that the interpreter will refer to when it tries to achieve goals.

A ‘program’ in Prolog is nothing more than a set of facts and rules which can be loaded into the database.

Facts

As the name suggests, facts represent assertions which the Prolog interpreter will regard as being true.

Examples of assertions:

– 'Bill is old'.
– 'Anne's age is 29'.
– 'John is the father of Mary'.
– 'Henry VIII was king of England from 1509 to 1547'.
– 'February has 28 days'.

Prolog has a way to express such assertions formally:

- old(bill).
- age(anne,29).
- father(john,mary).
- king(henry,8,england,1509,1547).
- month(february,28).
**Facts**

Suppose that we have in the Prolog database the fact

\[ \text{age(anne,29).} \]

Then the interpreter will respond Yes, given the following goal,

\[ ?- \text{age(anne,29).} \]

(this goal is immediately achieved):

\[ ?- \text{age(anne,29).} \]

More interestingly, if we give the goal

\[ ?- \text{age(anne,} A\text{).} \]

the interpreter will evaluate this by giving the value 29 to the variable A. And the answer will be

Yes.  A=29

?-

**Rules**

Besides simple facts, we can also store more complex information as rules. E.g.

- 'A person is old if he/she is over 70'.

- 'Date D1/M1 comes before date D2/M2 (in a given year)
  if M1 < M2 or (M1 = M2 and D1 < D2)'.

In Prolog:

\[ \text{old(P)} \leftarrow \text{age(P,A)}, \text{A} > 70. \]

\[ \text{earlier(D1,M1,D2,M2)} \leftarrow \]

\( M1 < M2 \); \( (M1 = M2, D1 < D2) \).

**Example Database**

\% A rule:

\[ \text{get_age} \leftarrow \]

\% Some facts:

\[ \text{age(jane,} 24\text{).} \]
\[ \text{age(jill,} 26\text{).} \]
\[ \text{age(julia,} 33\text{).} \]
\[ \text{age(mary,} 29\text{).} \]
\[ \text{age(alex,} 26\text{).} \]
\[ \text{age(arthur,} 26\text{).} \]
\[ \text{age(bill,} 33\text{).} \]
\[ \text{age(eric,} 17\text{).} \]
\[ \text{age(john,} 42\text{).} \]

\% Three important symbols in rules:

\% IF is represented by the symbol \( \leftarrow \)
\% AND is represented by a comma,
\% OR is represented by a semicolon

The head of a rule may not contain AND or OR symbols.

A rule has a head and a body, separated by the \( \leftarrow \) symbol.
Programs

Here is a simple Prolog program (expressed as a rule to be added to the database):

\[
\text{get_age :-} \\
\phantom{\text{get_age :-}} \text{write('Enter person: ')}, \\
\phantom{\text{get_age :-}} \text{read(P)}, \\
\phantom{\text{get_age :-}} \text{age(P,A)}, \\
\phantom{\text{get_age :-}} \text{write(P), write(' has age '), write(A)}. \\
\]

We 'run the program' by giving the goal

?- get_age.

The interpreter will scan its database and find a suitable rule, i.e. in this case a rule that "matches" the goal, here simply having the same head, i.e. defining the same concept. Then the problem reduces to solving the subgoals

\[
\text{write('Enter person: '), ..., age(P,A) ... .} \\
\]

If it doesn't, then \( \text{age(P,A)} \) will fail, and \text{get_age} will fail as well, with answer: \text{No}.

Example

Suppose we have a file (say called \text{ages.pl}) containing the age facts that we saw before. Let's suppose this file has been consulted (i.e. loaded in the database), by

?- consult(ages).

and consider some goals.

1. ?- age(eric,X). (succeeds, with \( X = 17 \)).
2. ?- age(alex,X). (succeeds, with \( X = 26 \)).
3. ?- age(susan,X). (fails).
4. ?- age(P,17). (succeeds, with \( P = \text{eric} \)).
5. ?- age(P,19). (fails).
6. ?- age(P,26). (succeeds, with \( P = \text{jill} \)).

Values for Variables

In all of the above, the interpreter achieves the goal by matching the goal with a fact in the database.

As seen, this matching process which gives values to variables.

- What if there are no matches in the database?
- What if there is a possible match in the database?
- What if there are more possible matches in the database?

We use 'Backtracking', more on this shortly...

Variables

What is a variable?
How is \( X \) a variable, but \( \text{mary} \) is the name of a person?

Important:

- There are no declarations of variables.
- Variables do not have types.
- It is sensible to choose names for variables which are suggestive of their intended use.
- Prolog is case-sensitive.
Variables

What is a variable?

How is $X$ a variable, but *mary* is the name of a person?

Important:

- **Assignment is destructive:** once assigned, the value of a specific variable is immediately propagated to all its occurrences, and the variable disappears (from the current computation!). It is not possible to assign it again. Compare this with $x = x + 1$
- **What scoping rules do we have?**
- **Variables in Prolog are not global.** In fact they behave **more like parameters** than variables.

Use Of Variables

To provide generality in rules, suppose that we have a collection of facts of the forms

- `father(james,mary).`
- `mother(jane,brian).`

Then it might be useful to have a rule:

```prolog
parent(X,Y) :-
  father(X,Y); mother(X,Y).
```

Here the $X$ and $Y$ stand for anything. Whatever $X$ and $Y$ stand for,
- $X$ is a parent of $Y$
- **IF**
- $X$ is the father of $Y$ or $X$ is the mother of $Y$

This use of variables is similar to the use of formal parameters in a method.

Use of Variables (2)

We use variables to pass values from one part of a computation to another:

```prolog
father_age(Person, Age) :-
  father(Father, Person),
  age(Father, Age),
  write('The father of '),
  write(Person),
  write(' has age '), write(Age).
```

We might then give a goal

```prolog
?- father_age(mary,A).
```

The variable Person is being used to carry a value **IN** to the operation of the rule. The variable Age is carrying a value **OUT**.

Backtracking

What happens in the previous example if the person entered is not known to the database (i.e. there is no father fact which can be used)?

- Answer: the goal fails, and the interpreter just answers No.

We can make this tidier by adding another rule:

```prolog
get_father_age :-
  write('Enter person: '),
  read(Person),
  father_age(Person,Age),
  write('The father of '),
  write(Person),
  write(' has age '), write(Age).
```

```prolog
get_father_age :-
  write('Not known.').
```
Backtracking (2)

The interpreter now has another way to achieve the goal

?- get_father_age.

Initially, it will try to use the first rule. If this fails, the interpreter will automatically backtrack to try to succeed another way by using the next matching rule it finds.

Failure always causes backtracking.

Backtracking (3)

Consider the age facts again: you want to write out a list of names of all the people with a given age.

```prolog
people_of_age(A) :-
    age(P,A),
    write(P), nl,
    fail.
people_of_age(A) :-
    write('End of list').
```

Two new things here: `nl` causes a new line on the output. `fail` (as a goal) always fails immediately.

So why make it fail?

This is our first example of a failure-driven loop.

– Note that this looping can be caused by any failure, not just by the explicit fail we have used.

Backtracking (4)

When the interpreter backtracks, it works its way back through all of the previous subgoals, trying to re-achieve each one. It is a general rule that read and write goals cannot be re-achieved, but here (possibly) `age(P,A)` can be.

When some subgoal has been re-achieved, the interpreter starts working forwards again from that point in the normal way, as if nothing had happened.

A subsequent failure will cause backtracking again.

We have used the explicit fail here, but any failure causes backtracking.

Note that, as a result of backtracking, variables can have their values changed (e.g. P above). This is the only way that values of variables can be changed.

Management of backtracking is fundamental to writing programs in Prolog.

Prolog compilers/interpreters

SWI Prolog

http://www.swi-prolog.org/

Swi-Prolog offers a comprehensive free Prolog environment. Since its start in 1987, SWI-Prolog development has been driven by the needs of real-world applications. SWI-Prolog is widely used in research and education as well as commercial applications. Join over a million users who have downloaded SWI-Prolog.

Download SWI-Prolog  Get Started  Is SWI-Prolog Right For My Project?
Recursion

Recursion in Prolog means placing in the body of a rule a call to the predicate which occurs in the head of the rule.

Here is a simple (bad) example:

\[
\text{stars} : - \text{write('**'), nl, stars}. \\
\]

You should try this (give the goal `?- stars.`).

It doesn't work properly, and it is important to see why (and to see what happens when you try it -- CTRL-C followed by 'a' will come in handy).

Recursion (2)

Recursion must always be made to terminate properly.

Here is a better example:

\[
\text{stars}(0) : - \text{nl}. \\
\text{stars}(N) : - \text{write('**'), nl, M is N-1, stars(M)}. \\
\]

And some others:

\[
\text{royal(victoria)}. \\
\text{royal(X) : - parent(P,X), royal(P)}. \\
\text{archer(dan)}. \\
\text{archer(X) : - father(P,X), archer(P)}. \\
\]

Lists

The most important data structure in Prolog is the list.

Actual lists are normally written out like this:

\[
[3,7,5,29,6,3,1,2] \\
[\text{mary, john, bill, arthur}] \\
[75] \\
[] \\
\]

But lists have alternative notations.

\[
[H|T] \quad \text{means the list with head H and tail T.} \\
[A,B|T] \quad \text{means the list whose first two members are A and B, with T as the rest of the list.} \\
\]
Lists (2)

In the examples above:

- 3 is the head and [7,5,29,6,3,1,2] is the tail.
- mary is the head and [john,bill,arthur] is the tail.
- 75 is the head and [] is the tail.

The empty list [] does not have a head and a tail.

[2,1,3]
[2|[1,3]]
[2,[1|[3|[]]]]

all represent the same list.

Built-In List Operations

Membership of a list: member(X,Y).
Succeeds when X belongs to list Y.

Length of a list: length(X,Y).
'Y is the length of list X'.

Concatenate lists: append(X,Y,Z).
'Z is the result of concatenating lists X and Y.'

Deleting from a list: delete(X,Y,Z).
'Z is the result of deleting all occurrences of Y from the list X.'

Last in a list: last(X,Y).
'X is the last member of list Y'.

Lists (3)

Notes:

- The tail of a list is a list.
- We must be careful to distinguish a one-member list from the object which is its single member.
- Processing lists is almost always done using recursion.
- Task: to write out the members of a list, each on a separate line.

\[
\text{writelist([], \text{write(H), nl, writelist(T).}})
\]

This is the first illustration of the use of patterns in the head of a rule. There are two cases here, represented by two rules, one for an empty list, and one for a list which is not empty. We shall see more of this, as it is significant.

Functions

Prolog does not allow functions (methods, procedures ...).

If we need to define a function to carry out certain computations, we can do so, but it must be done indirectly by means of a predicate. For example:

\[
\text{double(X,Y) :- Y is 2*X.}
\]

- Think of X and Y as parameters: X carries a value IN to the operation of double, and Y carries a value OUT.
- Example:

\[
\text{?- double(7,K).}
\]

- This will result in K being given the value 14.
Functions (2)

Compute the larger of two numbers:

\[
\text{max}(A,B,A) :\text{ } A \geq B.
\]
\[
\text{max}(A,B,B) :\text{ } B > A.
\]

The first two parameters carry inputs, the third carries the output.

Note that there is no explicit IF .. ELSE to distinguish the two cases. Prolog will use the first rule if it can, otherwise it will use the second.

Example goal:
? - max(4,7,M).

This will result in M being given the value 7.

Functions (3)

Another:

\[
\text{sumlist}([H|T],S) :\text{ -}
\]
\[
\text{sumlist}(T,N),
\]
\[
S \text{ is } H+N.
\]
\[
\text{sumlist}([X],X).
\]

This predicate will compute (as the value given to the second parameter) the sum of the numbers in a list (the first parameter).

Note the use of patterns for the IN parameter. The first clause deals with lists with one member only, the second deals with lists with more than one member.

Example goal:
? - sumlist([6,2,5,2],V).

This will result in V being given the value 15.

Functions (4)

And patterns can be used for OUT parameters, too:

\[
\text{duplicate}([],[]).
\]
\[
\text{duplicate}([H|T],[H,H|T]) :\text{ -}
\]
\[
\text{duplicate}(T,T1).
\]

This predicate will take a list (its first parameter), and duplicate every member of it, to give a new list as the value of the second parameter.

Example goal:
? - duplicate([6,2,5],E).

This will result in E being given the value [6,6,2,2,5,5].

Structure of Programs

How can a Prolog program be large and complex?
– Answer: rules for predicates can call other predicates.

Schematically, here is a more complicated program:

\[
a :\text{ - } b, \text{ c, d}.
\]
\[
b.
\]
\[
c :\text{ - } e, \text{ f}.
\]
\[
d :\text{ - } g.
\]
\[
e :\text{ - } h.
\]
\[
g.
\]
\[
h.
\]
Structure of Programs (2)

A program is a collection of facts and rules. Normally there will be a top level predicate — the one which will form the goal which is given to the interpreter when we 'run the program'.

It is standard practice to place the top-level predicate at the beginning of the program. In the above, a is the top-level predicate.

Programs can be spread over a number of files without difficulty. Before being run, a program must be in the database. It is loaded by using consult, and we may consult several files if we need to.

Achieving Goals

The interpreter is given a goal, it uses the first fact or rule in the database which matches the goal.

– This matching may give values to variables.

If a fact is used, the goal is achieved immediately.

If a rule is used, the components of the body of the rule become subgoals, which are individually dealt with, in order.

– Subgoals separated by commas must all be achieved.
– Subgoals separated by semicolons are alternatives (if one fails then the next is tried).

If any stage causes backtracking, the interpreter tries to find other ways to achieve goals.

Readability

It is very easy to write programs which are correct but incomprehensible.

Comments are essential in Prolog programs, and there should always be comments to indicate, for each predicate, which other predicates call it.

In the body of a rule, place the separate subgoals on separate lines.

Use of blank lines, or of comment lines consisting of asterisks (say) can also help. Avoid clutter.

Mixing up commas and semicolons in the body of a rule is a bad idea. It makes programs hard to follow, and you should avoid doing it if possible.

Example

Recall

\[
\text{parent}(X,Y) :- \text{father}(X,Y); \text{mother}(X,Y).
\]

Entirely equivalent to this is the following:

\[
\begin{align*}
\text{parent}(X,Y) & :- \text{father}(X,Y). \\
\text{parent}(X,Y) & :- \text{mother}(X,Y).
\end{align*}
\]

I.e. two alternative rules.

Frequently this can be done in preference to using a semicolon, and it makes programs much clearer.
Analysing the syntax of Prolog is useful to precisely identify what constitutes a legal expression, i.e. an expression which the Prolog interpreter will understand.

The syntax of Prolog is very simple, and may be expressed by the diagram:

```
    term
   /   \
variable  structure
   |    /   \
number  atom
```

Every legal expression is a term.

A term is either a constant, a variable, or a structure.

A constant is either a number or an atom.

Now we need only describe what numbers, atoms, variables and structures are.

A number is just what you think.
- Integers and floating-point numbers

An atom consists of characters in one of the following ways:
- beginning with a lower-case letter (internal underscores allowed).
- alphanumeric, in particular combinations which are known to the interpreter, e.g. `:-`, `=`, `>=`, etc.
- Any string of characters enclosed in single quotes. If you need to include a single quote in such an atom, put two single quotes, e.g. `'Bill''s House'`

A variable is an alphanumeric string which begins with an upper-case letter or an underscore.

A structure has a functor and arguments. The functor must be an atom, and the arguments can be Prolog terms of any kind. E.g.

```
child(mary,john)
date(29,2,96)
earlier(date(29,2,96),date(1,3,96))
```

Every legal expression in Prolog which is not a number, an atom or a variable, is a structure with a functor and arguments.
Forms of Structures

Some structures have alternative forms:

- Arithmetic expressions
- Lists
- Facts and rules

Arithmetic expressions:

\[
\begin{align*}
2 + 3 & \text{ means } +(2,3) \\
2 < 3 & \text{ means } <(2,3) \\
2 = 3 & \text{ means } = (2,3) \\
X \text{ is } 2 + 3 & \text{ means } is(X,+(2,3))
\end{align*}
\]

Forms of Structures

Lists (the functor is always .)

\[
\begin{align*}
[H|T] & \text{ means } .(H,T) \\
[2] & \text{ means } .(2,[]) \\
[2,5,5] & \text{ means } .(2,(5,(5,[])))
\end{align*}
\]

Facts and rules (the functor is always :-)

\[
\begin{align*}
\text{child}(X,Y) :- \text{parent}(Y,X).
\end{align*}
\]

is the same as

\[
\begin{align*}
\text{:-} & \text{(child}(X,Y),\text{parent}(Y,X))
\end{align*}
\]

age(anne,29).

is the same as

\[
\begin{align*}
\text{:-} & \text{(age}(anne,29),true)
\end{align*}
\]

Expressions in Prolog

Prolog can read/write an entire term in one operation.

There are no types in Prolog. Each variable can have as its value any Prolog term.

Numbers and structures are different things. The goal

\[
?- 2 + 3 = 5.
\]

will fail, because \(2 + 3\) is a structure, whereas 5 is a number.

To evaluate an arithmetic expression we can use is:

\[
?- X \text{ is } 2 + 3, X = 5.
\]

will succeed.

There is also :=:, which evaluates then tests equality:

\[
?- 2 + 3 =: 7 - 2.
\]

will succeed.

Data Structures

Strictly, the only kind of expression which can represent structured data is the structure. There are no arrays or pointers. We do have lists, of course (which are a special kind of structure).

Simple structures, e.g.

\[
\begin{align*}
\text{father}(\text{dan},\text{phil}) \\
\text{king}(\text{henry},8,\text{england},1509,1547) \\
\text{date}(\text{thursday},15,\text{february},1996) \\
\text{book}('\text{Algorithms}','\text{Sedgwick,R}',\text{Addison-Wesley}',1988)
\end{align*}
\]

These are structures which may be considered analogous to records. Such structures may be stored as facts, and data may be extracted by the use of goals containing variables:

\[
\begin{align*}
?- \text{king}(\text{henry},8,\text{england},X,Y).
\end{align*}
\]

\[
\begin{align*}
X & = 1509 \\
Y & = 1547
\end{align*}
\]
Summary

What we’ve covered:
- logic and argument
- goals and how to achieve them
- facts and rules
- the Prolog database
- predicates and programs
- use of variables
- backtracking
- recursion
- data structures including lists
- Prolog syntax

Prolog is different - it’s a different paradigm
- the idea of program is totally unfamiliar
- but some ideas persist from one paradigm to another