Logic Programming – Part I

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Suspended lectures

- Thursday April 11 (Provette)
- Tuesday April 16 (ICT days)
- No tutoring on Tuesday April 16 (ICT days)

- Next lecture Thu April 18

Today

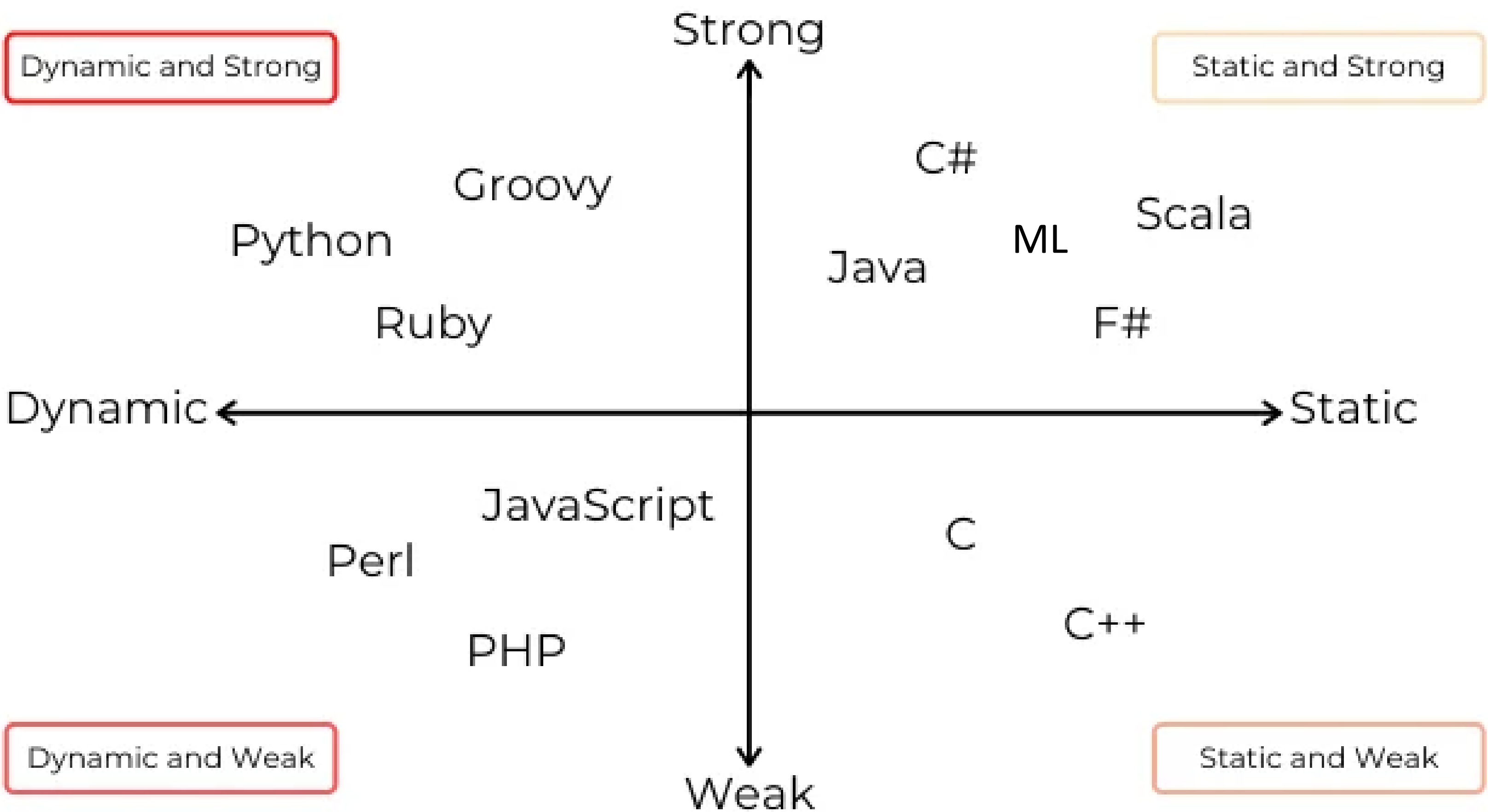
- Logic Programming
- Prolog
  - Syntax
  - Resolution and unification
  - Arithmetic
  - Functions
  - Lists

Recap

Type systems

- The type system of a language:
  1. Predefined types
  2. Mechanisms to define new types
  3. Control mechanisms
     - Equivalence
     - Compatibility
     - Inference
  4. specification of whether types are statically or dynamically checked

Strong, weak, dynamic and static



Strong typing is an aspect of type safety

[Prepinsta.com]

Type equivalence

- Two types `T` and `S` are equivalent if every object of type `T` is also of type `S`, and vice versa
- Two rules for type equivalence
  - Equivalence by name: the definition of a type is opaque
    - Strict
    - Loose
  - Structural equivalence: the definition is transparent

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

```
type T1 = int;
type T2 = char;
type T3 = struct{
    T1 a;
    T2 b;
}
type T4 = struct{
    int a;
    char b;
}
```

Type compatibility

- `T` is compatible with `S` if objects of type `T` can be used in contexts where objects of type `S` are expected
- Example: `int n; float r; r=r+n` in some languages
- In many languages compatibility is used for checking the correctness of:
  - Assignments (right-hand type compatible with left-hand),
  - parameter passing (actual parameter type compatible with formal one), …
- Compatibility is reflexive and transitive but it is not symmetric
  - E.g., compatibility between `int` and `float` but not viceversa in some languages

Type conversion

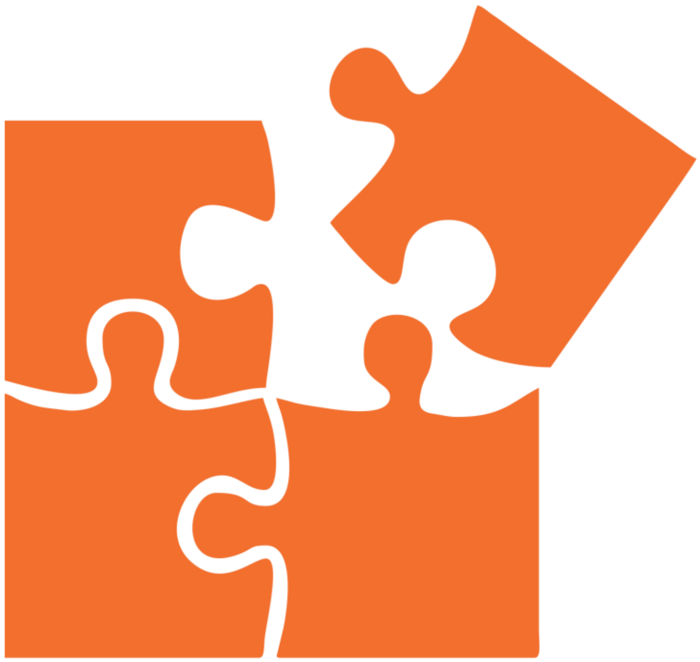
- If `T` is compatible with `S`, there is some type conversion mechanism.
- The main ones are:
  - Implicit conversion, also called coercion. The language implementation does the conversion, with no mention at the language level
  - Explicit conversion, or cast, when the conversion is mentioned in the program

Polymorphism

- A single value has multiple types
- Three forms of polymorphism
  - Ad hoc polymorphism (overloading) – e.g., + used with real, integers, …
  - Universal polymorphism
    - Parametric polymorphism (explicit or implicit)
      - Explicit: explicit annotation (<T>) indicating the types to be considered as parameters (e.g., C++ *template* and Java *generic*)
      - Implicit: the type checker tries to determine for each object the most general type from which the others can be obtained (e.g., ML)
    - Subtype or inclusion polymorphism

Abstract Data Types

- One of the major contributions of the 1970s
- Basic idea: separate the interface from the implementation
  - Interface: types and operations that are accessible to the user
  - Implementation: internal data structures and operations acting on the data types
  - Example
    - Sets have operations as `empty`, `union`, `insert`, `is_member?`
    - Sets can be implemented as vectors, lists etc.



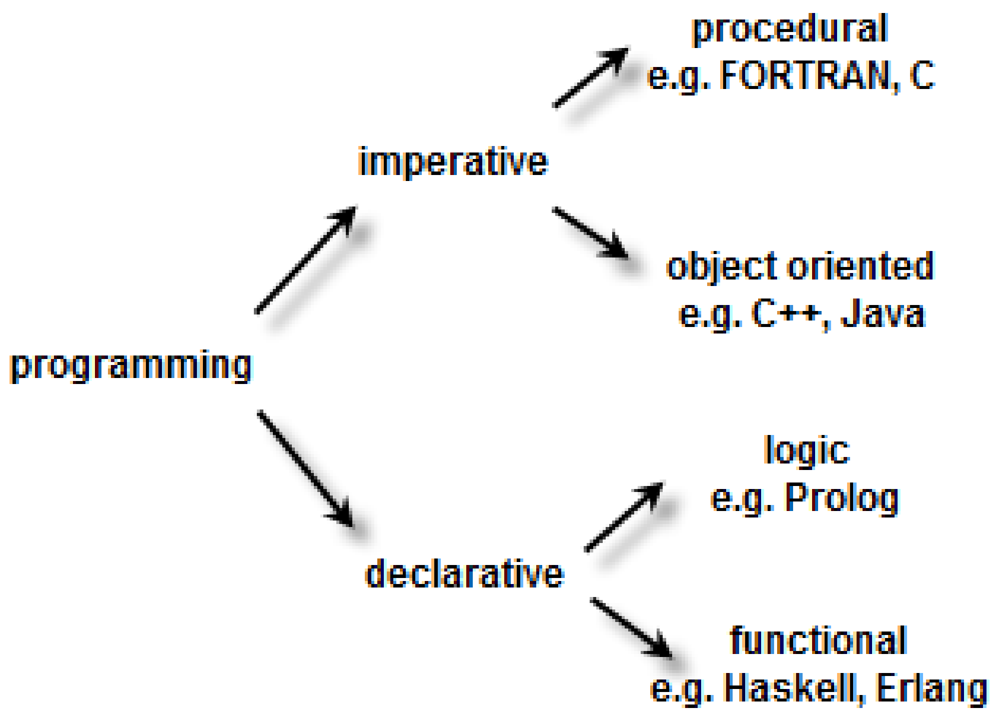
Logic Programming

Different characteristics

- Imperative (how to do?)
  - Specify a sequence of operations that modify a state (statements)

- Declarative (what to do?)
  - What needs to be solved to get the result

… different languages



… different languages

- Imperative (how to do?)
  - Classical: Fortran, Pascal, C
  - Object-oriented: Smalltalk, C++, Java
  - Scripting: Perl, Python, Javascript

- Declarative (what to do?)

```
int main(){
    printf("Hello World");
    return 0;
}
```

```
public class HelloWorld{
  public static void
main(String[] args) {
    System.out.println("He
llo World"); }}
```

```
print ‘‘Hello, world!\n’’
```

… different languages

- Imperative (how to do?)
  - Classical: Fortran, Pascal, C
  - Object-oriented: Smalltalk, C++, Java
  - Scripting: Perl, Python, Javascript

- Declarative (what to do?)
  - Functional: ML, Ocaml
  - Logic: Prolog

```
output = program (input)
```

```
program (input, output)
```

Logic Programming Concepts

- The programmer states a collection of axioms from which theorems can be proven
- The programmer states a goal, and the language implementation attempts to find a collection of axioms and inference steps (including choices of values for variables) that together imply the goal
- Prolog is the most widely used such language

Logic Programming Concepts

- In most logic languages, axioms are written in a standard form known as a Horn clause
- A Horn clause consists of a head, or consequent term `H`, and a body consisting of terms `Bi`
- We write

```
H:- B1, B2 , ... , Bn
```

- The semantics of this statement are that when the `Bi` are all true, we can deduce that `H` is true as well
- We can read this as "`H`, if `B1, B2, ...`, and `Bn`"
- Horn clauses can be used to capture most, but not all, logical statements

Horn clauses

- In order to derive new statements, a logic programming system combines existing statements, through a process known as resolution
  - If we know that A and B imply C, and that C implies D, we can deduce that A and B imply D
  - Terms such as A, B, C, and D may consist not only of constants, but also predicates applied to atoms or to variables
  - During resolution, free variables may acquire values through unification with expressions in matching terms

```
C :- A, B;
D :- C;
----------
D:- A,B
```

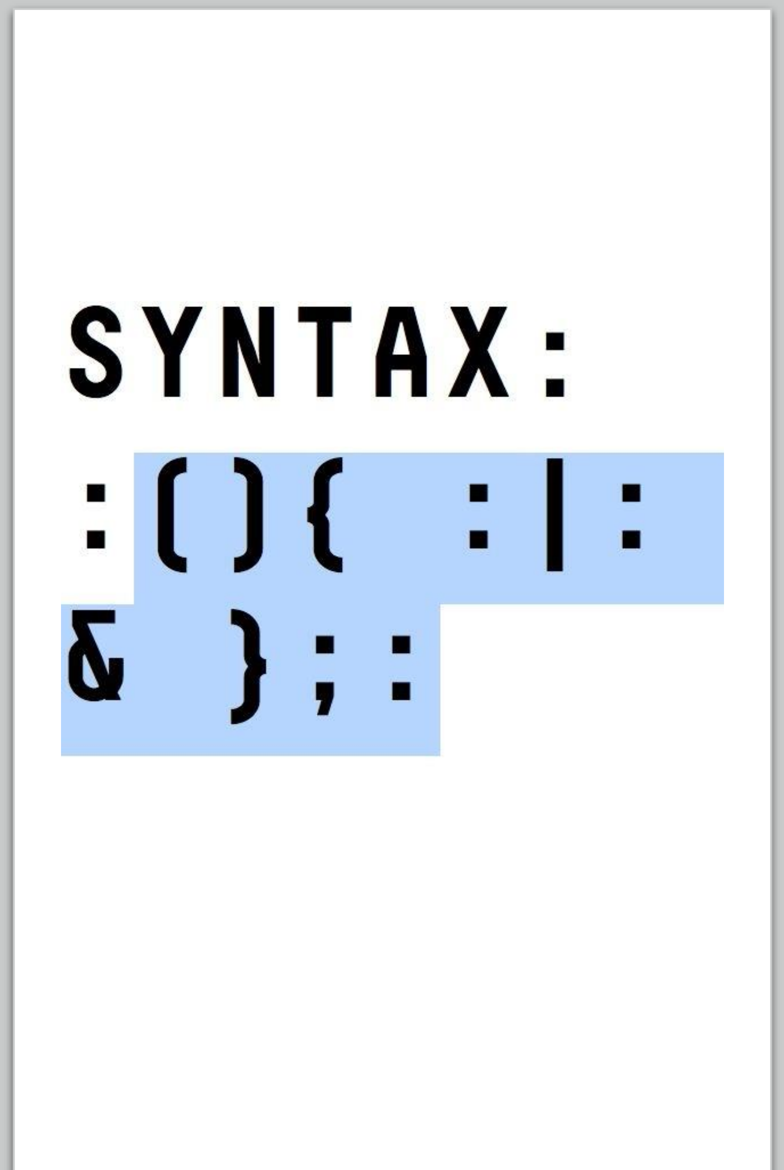
```
p(X) :- q(X);
q(1);
-----------------
p(1)
```

```
rainy(rochester).
rainy(seattle).
cold(seattle).
```

```
snowy(X):-rainy(X),cold(X).
```

```
?- son(X,Y).
X = charlie, Y = bob;
```

Prolog



Syntax

Prolog

- A Prolog interpreter runs in the context of a database of clauses (Horn clauses) that are assumed to be true
- Each clause is composed of terms
- A term may be:
  - a constant
  - a variable
  - a structure

Prolog

- A term may be:
  - a constant may be an atom or a number
    - An atom: looks like an identifier beginning with a lowercase letter, a sequence of “punctuation” characters, or a quoted character string
    - A number
  - a variable: like an identifier beginning with an uppercase letter
    - Variables can be instantiated to (i.e., can take on) arbitrary values at run time as a result of unification
  - structure:
    - a logical predicate or
    - a data structure

```
bob horse2
'horse' mario
...
```

```
123 -234
3.14
```

```
X AA List ...
```

```
sum(2,3)
bigger(horse,duck)
...
```

Structures

- Structures consist of an atom, called the functor, and a list of arguments

```
teaches(scott,cs254)
bin_tree(foo,bin_tree(bar,glarch))
```

functor arguments

- Prolog requires the opening parenthesis to come immediately after the functor, with no intervening space
- Arguments can be arbitrary terms: constants, variables, or (nested) structures
- Conceptually, the programmer may think of certain structures as logical predicates
- We use the term predicate to refer to the combination of a functor and an "arity" (number of arguments)

Clauses: facts and rules

- The clauses in a Prolog database can be classified as
  - facts or
  - rules
- Both end with a period
- A fact is a Horn clause without a right-hand side
- Thus it looks like this (the implication symbol is implicit)

```
rainy(rochester).
```

- A fact can be expressed as `p(t1,...,tn)` where `p` is the name of the fact and `t1, ..., tn` are terms

Facts and rules

- A rule has a right-hand side:

  `snowy(X):-rainy(X),cold(X).`
- The token :- is the implication symbol, and the comma indicates “and”
- `X` is snowy if `X` is rainy and `X` is cold
- A program is a sequence of clauses

An example of Prolog program

lowercase for atoms

```
% facts:
rainy(rochester).
rainy(seattle).
cold(rochester).

% rules for "X is snowy"
snowy(X):-rainy(X),cold(X).
```

uppercase for variables

Clauses:
Fact and rules

Query (or goal)

- Goal: the predicate we wish to prove to be true.
- Clause with an empty left-hand side
  - Queries do not appear in Prolog programs
  - Rather, one builds a database of facts and rules and then initiates execution by giving the Prolog interpreter (or the compiled Prolog program) a query to be answered (i.e., a goal to be proven)
  - In most implementations of Prolog, queries are entered with a special ?− version of the implication symbol

Asking for a query (goal)

- Typing the following:

```
rainy(seattle).
rainy(rochester).
?- rainy(C).
```

the Prolog interpreter would respond with `C = seattle`.

- Of course, `C = rochester` would also be a valid answer, but Prolog will find `seattle` first, because it comes first in the database
  - One of the differences between Prolog and pure logic

More solutions for a query (goal)

- To find all possible solutions, we can ask the interpreter to continue by typing a semicolon:

```
C = seattle;
C = rochester
```

- With another semicolon, the interpreter will indicate that no further solutions are possible:

```
C = seattle;
C = rochester;
False.
```

- Given

```
rainy(seattle).
rainy(rochester).
cold(rochester).
snowy(X):-rainy(X),cold(X).
```

  the query `?- snowy(C)` yields only one solution:

```
C=Rochester.
```

Question 1

- Facts

```
mouse(mickey).
mouse(jerry).
cat(tom).
cat(felix).
```

- Query

```
?- cat(X).
```

```
A.X = jerry.
B.X = tom.
C.X= tom; X=felix.
D.false.
```

Answer question 1

- Facts

```
mouse(mickey).
mouse(jerry).
cat(tom).
cat(felix).
```

- Query

```
?- cat(X).
```

```
A.X = jerry.
B.X = tom.
C.X= tom; X=felix.
D.false.
```

Question 2

- Facts

```
mouse(mickey).
mouse(jerry).
cat(tom).
cat(felix).
taller(tom,felix).
taller(X,Y):-
    cat(X), mouse(Y)
```

- Query

```
?- taller(tom, X).
```

```
A.X=jerry.
B.X=jerry; X=mickey.
C.X=tom; X=felix.
D.X=felix; X=mickey;
  X=jerry.
```

Answer question 2

- Facts

```
mouse(mickey).
mouse(jerry).
cat(tom).
cat(felix).
taller(tom,felix).
taller(X,Y):-
    cat(X), mouse(Y)
```

- Query

```
?- taller(tom, Z).
```

```
A. X=jerry.
B. X=jerry; X=mickey.
C. X=tom; X=felix.
D. X=felix; X=mickey;
   X=jerry.
```

Declarative and procedural interpretation

- A clause can be interpreted
  - in a declarative way
    - *H :- B_1, …, B_n* If B_1, …, B_n are true, then also H is true
    - A query is a formula for which we want to prove that is a logical consequence of the program
  - In a procedural way
    - *H :- B_1, …, B_n* To prove/compute H it is necessary first to prove/compute B_1, …, B_n
    - A predicate is the name of a procedure, whose defining clauses constitute the body
    - The goal is a sort of main

If you are curious to try …

- You can try with SWI-Prolog

  https://www.swi-prolog.org/
- `?- [‘namefile.pl’].`
- Where namefile.pl contains facts and rules, while the goal is inserted via prompt. Alternatively:
- `?- consult(namefile).`

- If you want to edit and then reload:

```
?- edit(filename).
?- make.
```

- To exit:

```
?- halt.
```

Resolution and unification

Resolution

- The resolution principle (Robinson) says that if `C1` and `C2` are Horn clauses and the head of `C1` matches one of the terms in the body of `C2`, then we can replace the term in `C2` with the body of `C1`

Resolution

- Consider the following

```
takes(alice, his201).
takes(alice, cs254).
takes(bob, art302).
takes(bob, cs254).
classmates(X, Y) :- takes(X, Z), takes(Y, Z).
```

- If we let `X` be `alice` and `Z` be `cs254`, we can replace the first term on the right-hand side of the last clause with the (empty) body of the second clause, yielding the new rule

```
classmates(alice, Y) :- takes(Y, cs254).
```

- In other words, `Y` is a classmate of `alice` if `Y` takes `cs254`.

Unification

- The pattern-matching process used to associate `X` with `alice` and `Z` with `cs254` is known as unification
- Variables that are given values as a result of unification are said to be instantiated

Unification in Prolog

```
takes(alice, cs254)
takes(bob, cs254)
```

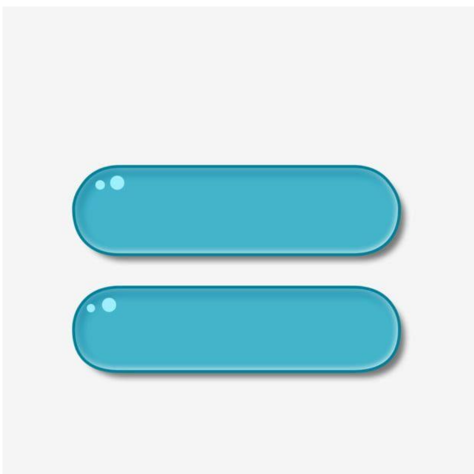
- Unification is a key feature in Prolog
- Two terms unify
  - if they are identical

    `?- takes(alice, cs254).` -> unifies directly with the fact
  - they can be made identical by substituting variables

    `?- takes(alice, X).` -> variable `X` is instantiated with `cs254`
- The idea is unifying the goal with the head of a rule
  - If succeeds, clauses in body become subgoals
  - Continue until all subgoals are satisfied
    - If search fails, backtrack and try untried subgoals

Unification in Prolog

- The unification rules for Prolog are as follows:
  - A constant unifies only with itself
  - Two structures unify if and only if they have the same functor and the same number of arguments, and the corresponding arguments unify recursively
  - A variable unifies with anything
    - If the other thing has a value, then the variable is instantiated
    - If the other thing is an uninstantiated variable, then the two variables are associated in such a way that if either is given a value later, that value will be shared by both.



Equality

Equality in Prolog

- Equality in Prolog is defined in terms of "unifiability"
- The goal `?-A=B.` succeeds if and only if `A` and `B` can be unified

Example

```
?- a = a.
true. % constant unifies with itself

?- a = b.
false. % but not with another constant

?- foo(a, b) = foo(a, b).
true.

?- X = a.
X = a. % only one possibility

?- foo(a, b) = foo(X, b).
X = a. % arguments must unify only one possibility
```

Equality

- Two variables can be unified without instantiating them
- If we type

```
?- A = B.
```

the interpreter will respond

```
A = B.
```

Arithmetic

Arithmetic

- The usual arithmetic operators are available in Prolog, but they play the role of predicates, not of functions
- `+(2, 3)`, which may also be written `2 + 3`, is a two-argument structure, not a function call
- This means that it will not unify with 5

```
?- (2 + 3) = 5.
false.
```

Arithmetic

- To handle arithmetic, Prolog provides a built-in predicate, `is`, that unifies its first argument with the arithmetic value of its second argument

```
?- is(X, 1+2).
X=3.


?- X is 1+2.
X = 3.

?- 1+2 is 4-1.
false.

?- X is Y.
<error> Arguments are not sufficiently instantiated

?- Y is 1+2, X is Y.
Y=X, Y = 3.
```

Question 3

- Query

```
?- 5=3+2.
```

A. True.
B. False.

Answer question 3

- Query

```
?- 5=3+2.
```

```
A.True.
B.False.
```

Question 4

- Query

```
?- 5 is 3+2.
```

A. True.
B. False.

Answer question 4

- Query

```
?- 5 is 3+2.
```

A. True.

B. False.

Question 5

- Query

```
?- 4+1 is 3+2.
```

A. True.
B. False.

Answer question 5

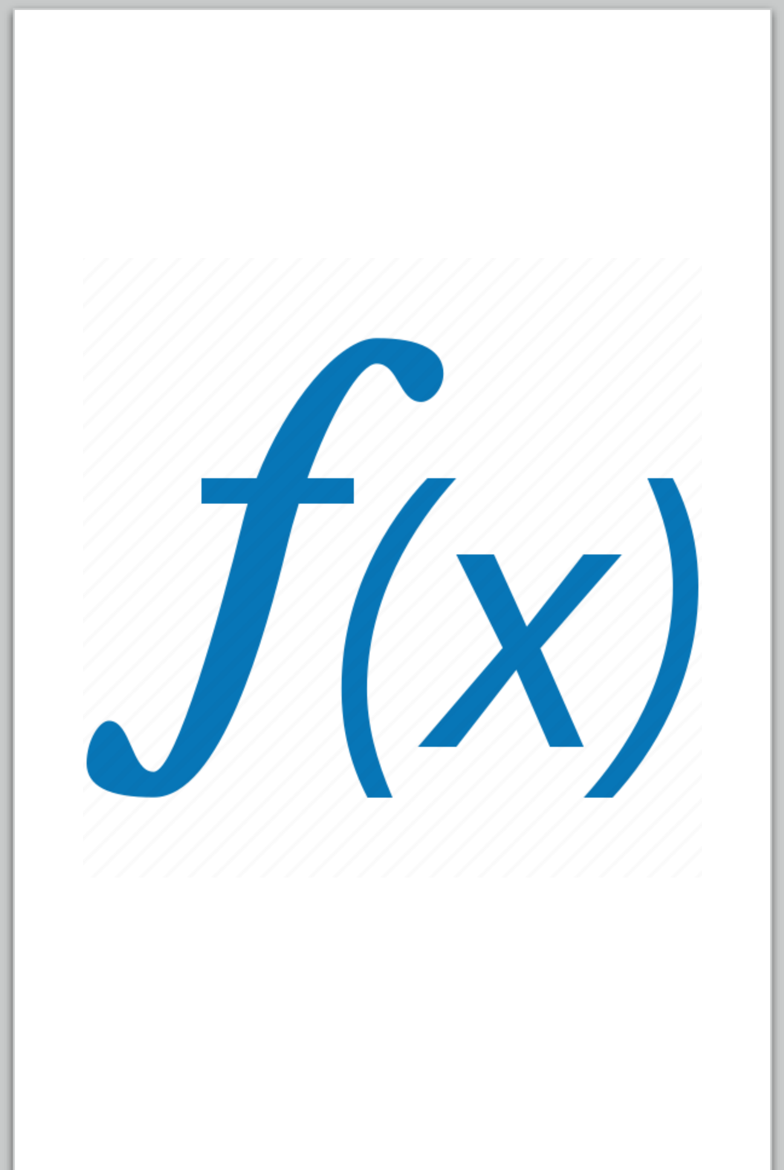
- Query

```
?- 4+1 is 3+2.
```

A. True.
B. False.

No mutable variables

- `=` and `is` operators do not perform assignment
  - Variables take on exactly one value ("unified")
- Example
  - `foo(...,X) :- ... X = 1,... % true only if X = 1`
  - `foo(...,X) :- ... X = 1, ..., X = 2, ... % always fails`
  - `foo(...,X) :- ... X is 1,... % true only if X = 1`
  - `foo(...,X) :- ... X is 1, ..., X is 2, ... % always fails: X can't be unified with 1 & 2 at the same time`



Functions

Function parameters and return value

- `increment(X,Y) :- Y is X+1.`
- `?-increment(1,Z).`
  `Z=2.`
- `?-increment (1,2).`
  `True.`
- `?-increment(Z,2).`
  `Arguments are not sufficiently instantiated.`

X+1 cannot be evaluated since X has not yet been instantiated.

- `addN(X,N,Y):- Y is X+N.`
- `?- addN(1,2,Z)`
  `Z = 3.`

Recursion

- `addN(X,0,X).`
-
```
addN(X,N,Y):- X1 is X+1,
              N1 is N-1,
              addN(X1,N1,Y).
```
- `?-addN(1,2,Z).`

`Z=3.`

addN is defined as recursively adding 1 to X N times

Question 6

- Facts

```
unknown(_,0,1).
unknown(X,1,X).
unknown(X,N,Y) :-
        N > 1,
        X1 is X*X,
        N1 is N-1,
         unknown(X1,N1,Y).
```

- Query

```
?- unkonwn(5,2,X).
```

A. X=1.
B. X=32.
C. X=25.
D. X=1; X=25.

Answer question 6

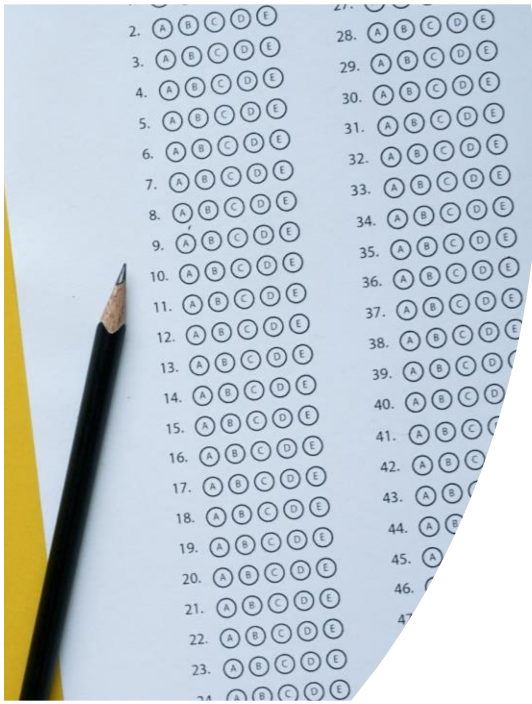
- Facts

```
unknown(_,0,1).
unknown(X,1,X).
unknown(X,N,Y) :-
        N > 1,
        X1 is X*X,
        N1 is N-1,
        unknown(X1,N1,Y).
```

- Query

```
?- unkonwn(5,2,X).
```

A. X=1.
B. X=32.
C. X=25.
D. X=1;X=25.



Lists

Lists

- A list is a finite sequence of elements
- List elements in Prolog are enclosed in square brackets
- Example: `[a,c,2,'hi',[W,3]]`
- The length of a list is the number of elements it has
- All sorts of Prolog terms can be elements of a list
- There is a special list: the empty list [ ]

Head and Tail

- A non-empty list can be thought of as consisting of two parts
  - The head
  - The tail
- As in ML, the head is the first item in the list
- The tail is everything else
  - The tail is the list that remains when we take the first element away
  - The tail of a list is always a list

Lists

- The construct `[a, b, c]` is shorthand for the compound structure `.(a, .(b, .(c, [])))`, where `[]` is an atom (the empty list) and . is a built-in cons-like predicate.
- How does it work matching?

```
?-[X,1,Z]=[a,_,17]
X=a,
Z=17
```

- `_` is the anonymous variable: used when we need a variable but we are not interested in how it is instantiated. Each occurrence is independent, i.e., it can be bound to something different

Vertical bar notation

- Prolog adds an extra convenience: an optional vertical bar that delimits the tail of the list
- Using this notation, `[a, b, c]` can be expressed as `[a | [b, c]]`, `[a, b | [c]]`, or `[a, b, c | []]`
- `[H|T]` is syntactically similar to ML `h::t`

```
?-[Head|Tail] = [a,b,c].
 Head = a.
 Tail = [b,c].
```

- The vertical bar notation is particularly useful when the tail of the list is a variable

Examples

- ?-[X,Y,Z]=[1,2,3].

  X=1.

  Y=2.

  Z=3.
- ?-[1,2,3,4]=[_,X|_].

  X = 2.
- ?-[1,2|X]=[1,2,3,4,5].

  X= [3,4,5].

Defining more complex predicates: `member` and `sorted`

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

sorted([]). % empty list is sorted
sorted([X]). % singleton is sorted
sorted([A, B | T]) :- A =< B, sorted([B | T]).
% compound list is sorted if first two elements
are in order and the remainder of the list
(after first element) is sorted
```

- Here =< is a built-in predicate that operates on numbers

append (or concatenate)

- append(L1,L2,L3) succeeds when L3 unifies with L2 appended at the end of L1, that is L3 is the concatenation of L1 and L2.
- Given this definition:

```
  append([], L2, L2). /*if L1 is empty,
then L3 = L2 */
  append([H | L1], L2, [H | L3]) :-
append(L1, L2, L3) /*prepending a new
element to L1, means prepending it to L3
as well*/
```

Examples

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

Readings

- Chapter 12 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill
- Few slides from the University of Maryland

Summary

- Logic Programming
- Prolog
  - Syntax
  - Resolution and unification
  - Arithmetic
  - Functions
  - Lists

Next time

- Logic Programming (second part)