b
UNIVERSITÄT
BERN

•

10. Logic Programming

Nataliia Stulova

•

Logic programming: why?

1960s-1970s: search for new approaches in knowledge *representation* and *reasoning* for AI

- representation: axioms in some logic
- reasoning: inference rules to prove new theorems from axioms

```
Example: first order logic

Axioms:  Oscar is a professor.

Inference rule: If A is a professor, then A gives lectures.

Theorem: Oscar gives lectures.
```

Logic programming: vison

Program consists of two parts:

- *database of facts* (axioms)
  - what data we have about problem domain
- *set of inference rules*
  - how to infer new data

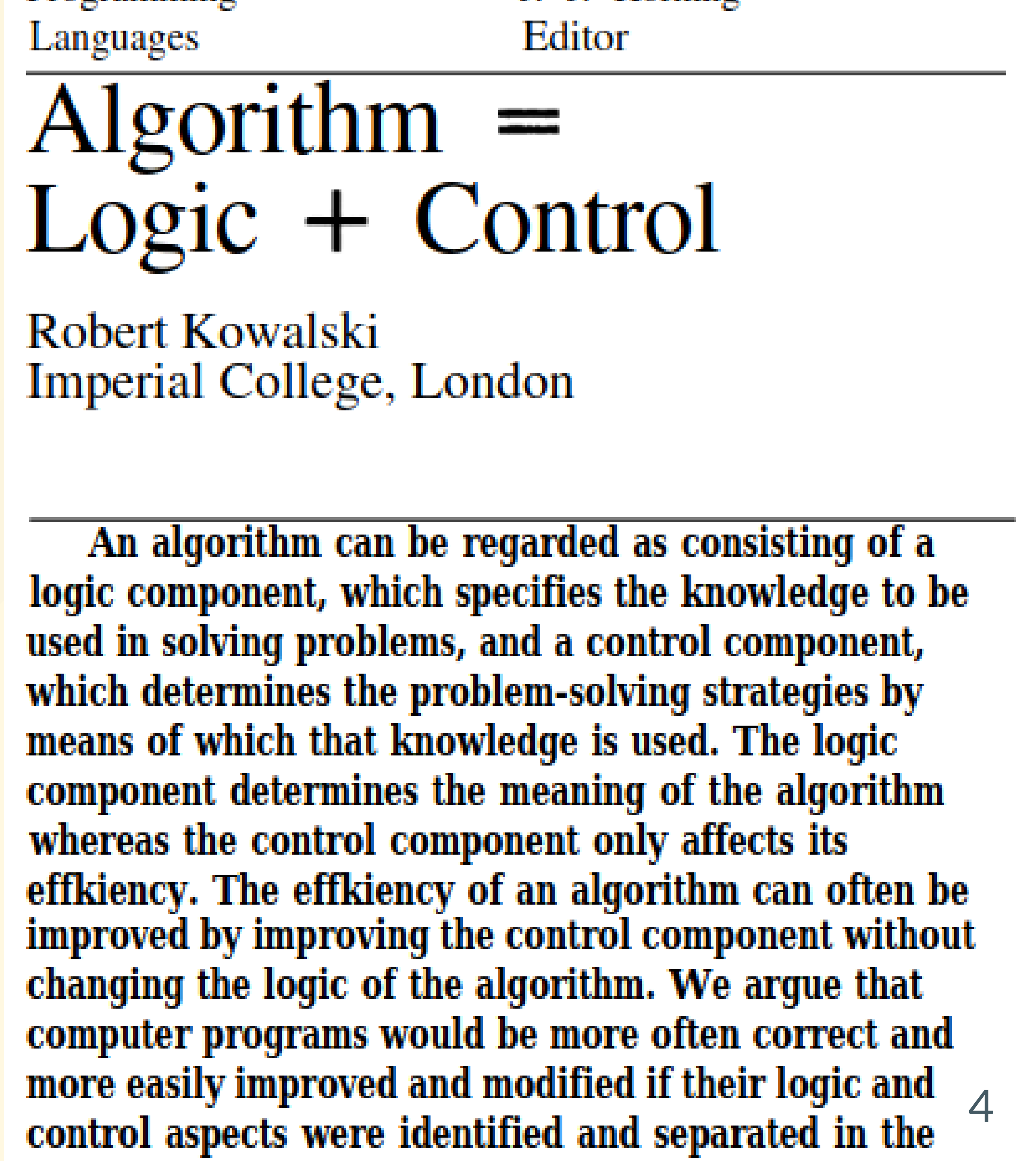
Different paradigms

Logic programming

Program = Facts + Rules

or

Program = Logic + Control



Languages Editor

Algorithm = Logic + Control

Robert Kowalski
Imperial College, London

An algorithm can be regarded as consisting of a logic component, which specifies the knowledge to be used in solving problems, and a control component, which determines the problem-solving strategies by means of which that knowledge is used. The logic component determines the meaning of the algorithm whereas the control component only affects its effkiency. The effkiency of an algorithm can often be improved by improving the control component without changing the logic of the algorithm. We argue that computer programs would be more often correct and more easily improved and modified if their logic and control aspects were identified and separated in the

Different paradigms

Logic programming

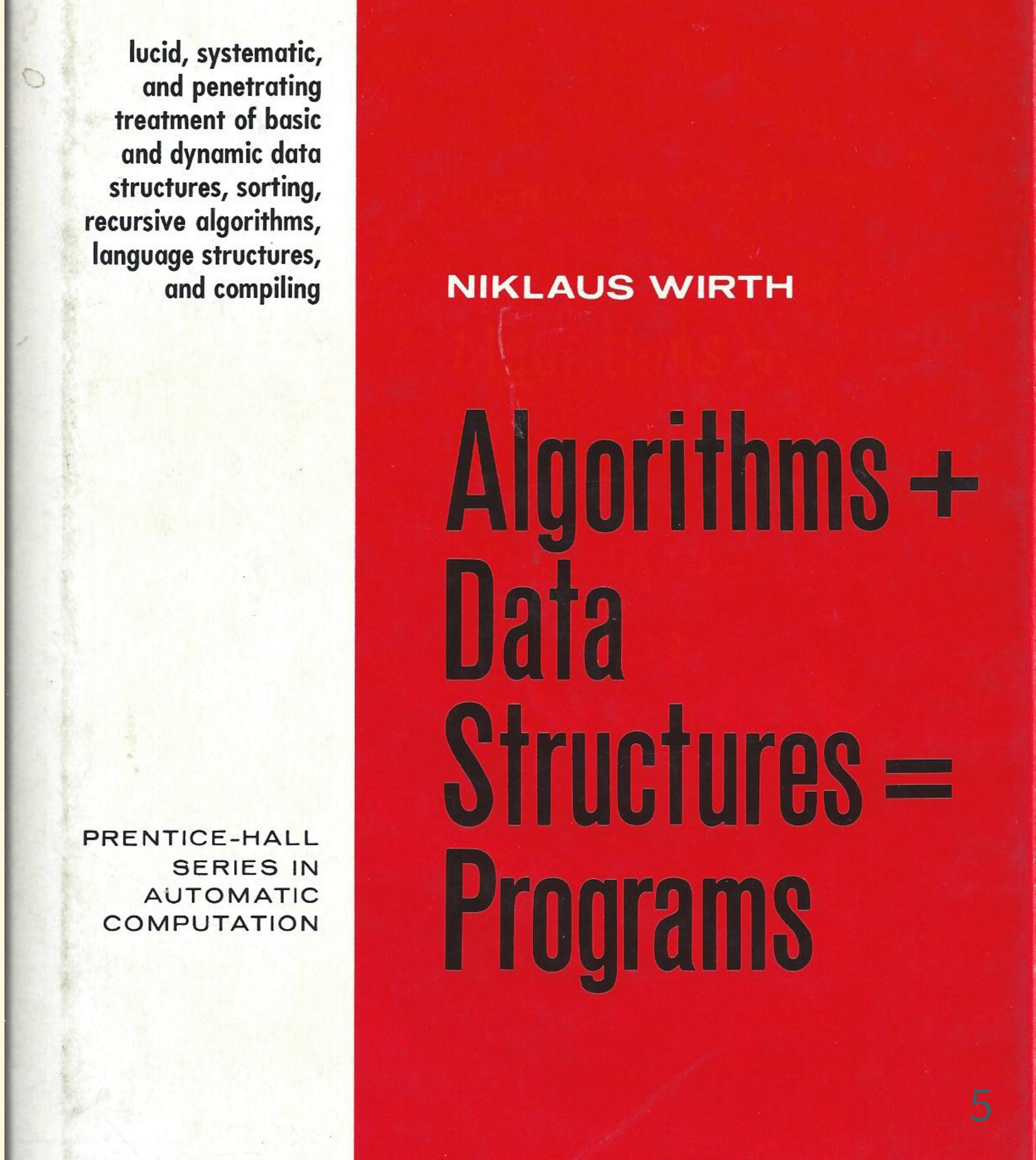
Program = Facts + Rules

or

Program = Logic + Control

Imperative programming

Program = Data structures + Algorithms



lucid, systematic, and penetrating treatment of basic and dynamic data structures, sorting, recursive algorithms, language structures, and compiling

NIKLAUS WIRTH

Algorithms + Data Structures = Programs

PRENTICE-HALL SERIES IN AUTOMATIC COMPUTATION

Logic programming: first steps

- 1969 PLANNER programming language (MIT, by Carl Hewitt)
- 1970 SHRDLU (MIT, by Terry Winograd): NL conversation program about block world

```
Person: Which cube is sitting on the table?
Computer: THE LARGE GREEN ONE WHICH SUPPORTS THE RED PYRAMID.
Person: Is there a large block behind a pyramid?
Computer: YES, THREE OF THEM: A LARGE RED ONE, A LARGE GREEN CUBE, AND THE BLUE ONE.
Person: Put a small one onto the green cube which supports a pyramid.
Computer: OK.
```

Logic programming: Prolog

- 1972 Prolog (Marseille, by Alain Colmerauer *et al.*)
  - enables programmers to write highly declarative programs that express intuitively logical queries,
  - uses a very powerful backtracking algorithm to answer those queries.

Prolog language basic syntax (1/3)

- variables (named or *anonymous*, like last one) `X`, `Value`, `A12`, `_42`, `Res`, `_`
- numbers `1`, `-15`, `3.1415`, `0.23e-5`
- constants (AKA atoms) `aa`, `[]`, `'hello'`, `@`
- logical connectives: `,` **and**, `;` **or**
- comments: `% full line` or `/* C-style inline */`

Prolog language basic syntax (2/3)

- compound terms:
  - can denote nested **data** structures `person(name(curt), age(27))`
  - can denote **predicates** (logical relations, coming next) and **functions** (relations between expressions and values, stay tuned)

Name of the compound term is called a *functor*.

To refer to a term we can write its functor and the number of arguments: `person/2`, `age/1` *etc.*

Prolog language basic syntax (3/3)

- facts (named **relations** between entities in the problem domain)

```
female(elizabeth).

% 'elizabeth' is a parent of 'charles'
parent(elisabeth, charles).                      % here ',' is just a comma
```

- rules (named **relations** that can be inferred from other relations)

```
% X is a mother of Y IF X is a parent of Y AND X is female
mother(X, Y) :- parent(X, Y), female(X).    % here ',' is a connective
```

Prolog programs are sets of **clauses**, that can be either.

Horn clauses

Prolog clauses `H :- B1, ..., Bn.`

are instances of *Horn clauses* `H if B0 and B1 and ... and Bn`

`H` is the **head** and `B0 and B0 and ... Bn` is the **body** of the clause.

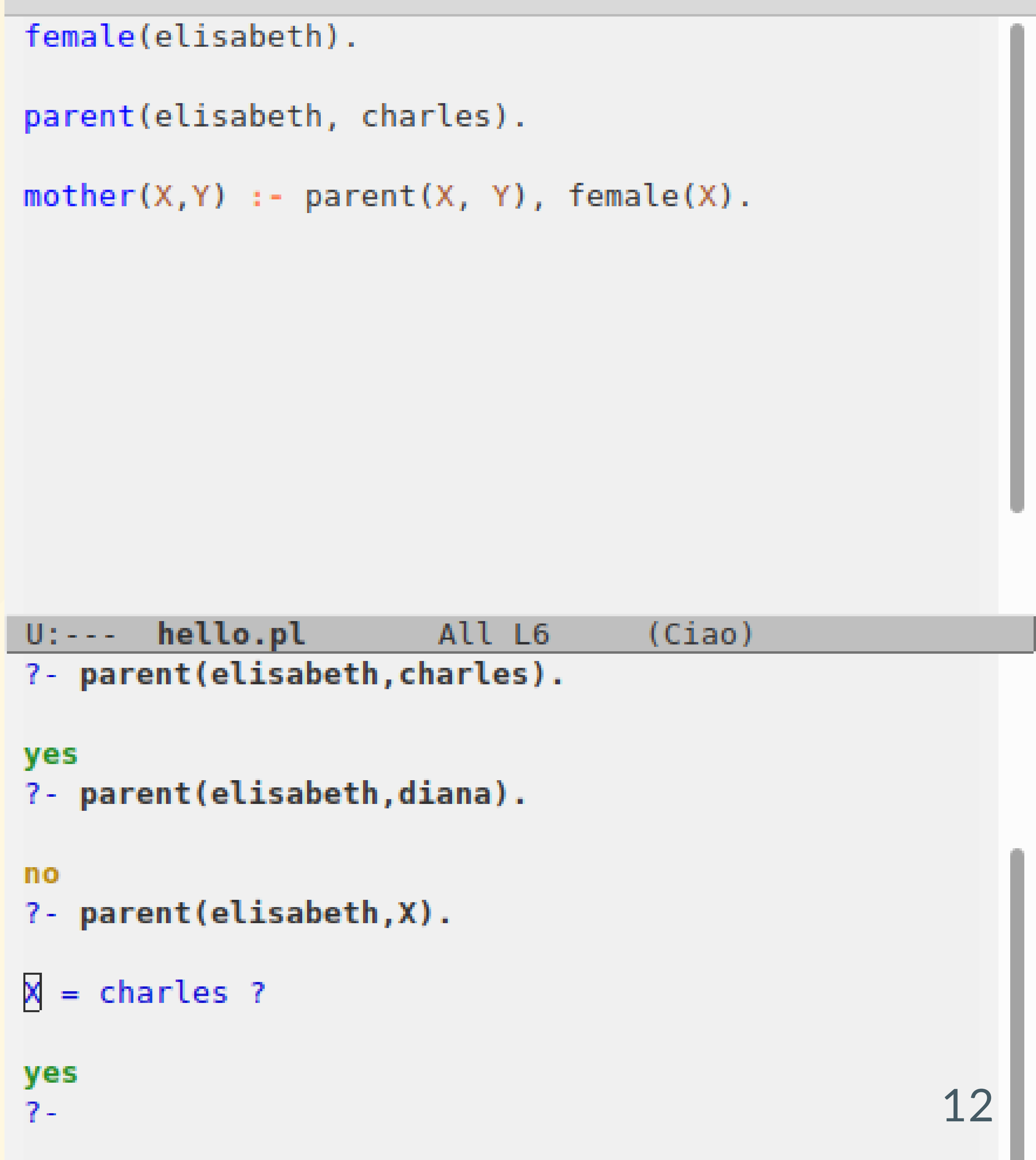
```
HEAD            BODY
mother(X,Y) IF parent(X,Y) AND female(X)  % rule
female(X)   IF true                       % fact
```

Computation in an interpreter

Prolog program execution starts with a **query** (AKA goal)

marked as `?-` in the interpreter window ->

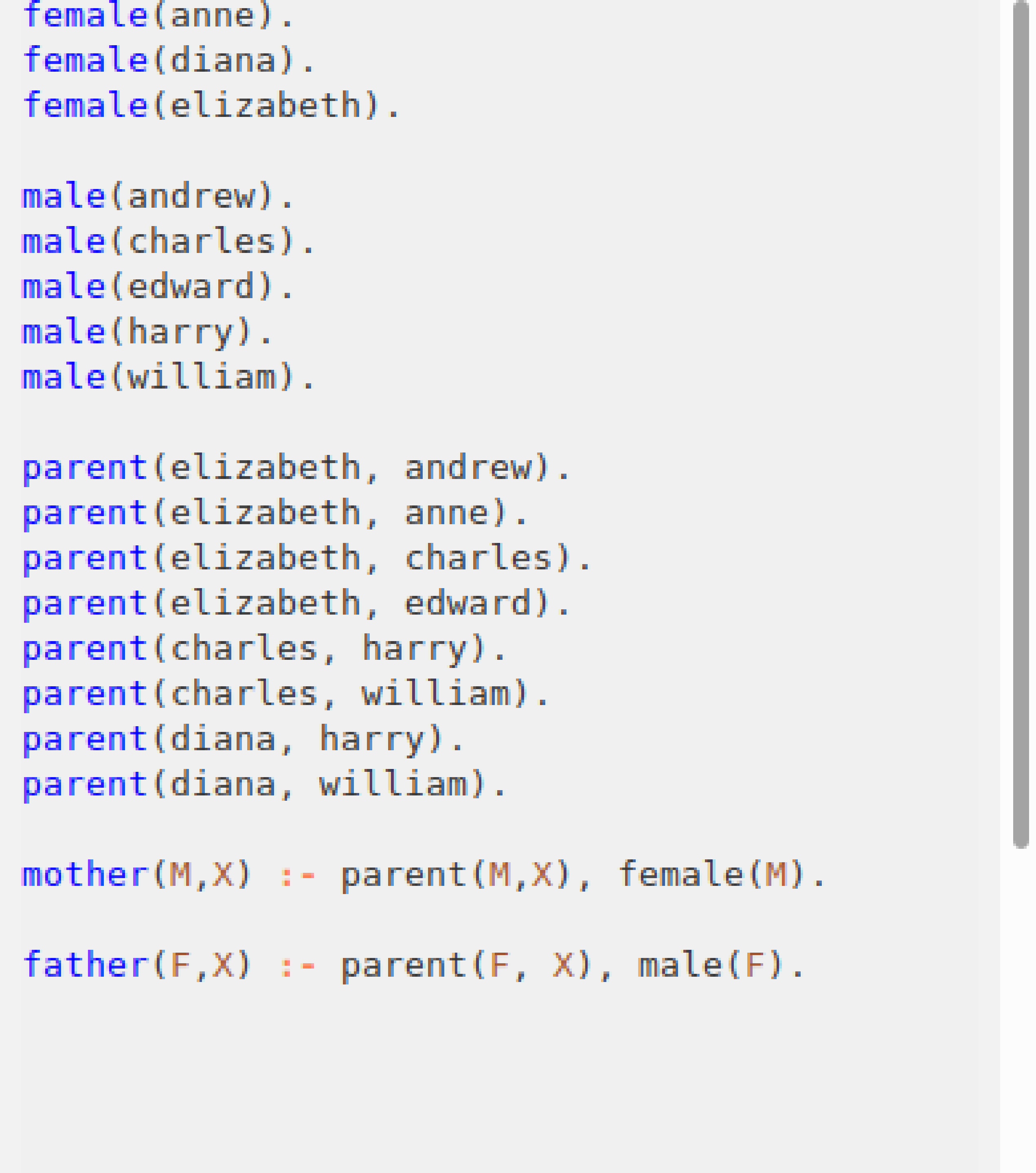
A query is a statement that can be answered by expansion using facs and rules.



```
female(elisabeth).

parent(elisabeth, charles).

mother(X,Y) :- parent(X, Y), female(X).
```

```
U:---  hello.pl       All L6      (Ciao)
```

```
?- parent(elisabeth,charles).

yes
?- parent(elisabeth,diana).

no
?- parent(elisabeth,X).

X = charles ?

yes
?-
```



```
female(anne).
female(diana).
female(elizabeth).

male(andrew).
male(charles).
male(edward).
male(harry).
male(william).

parent(elizabeth, andrew).
parent(elizabeth, anne).
parent(elizabeth, charles).
parent(elizabeth, edward).
parent(charles, harry).
parent(charles, william).
parent(diana, harry).
parent(diana, william).

mother(M,X) :- parent(M,X), female(M).

father(F,X) :- parent(F, X), male(F).
```

Running example

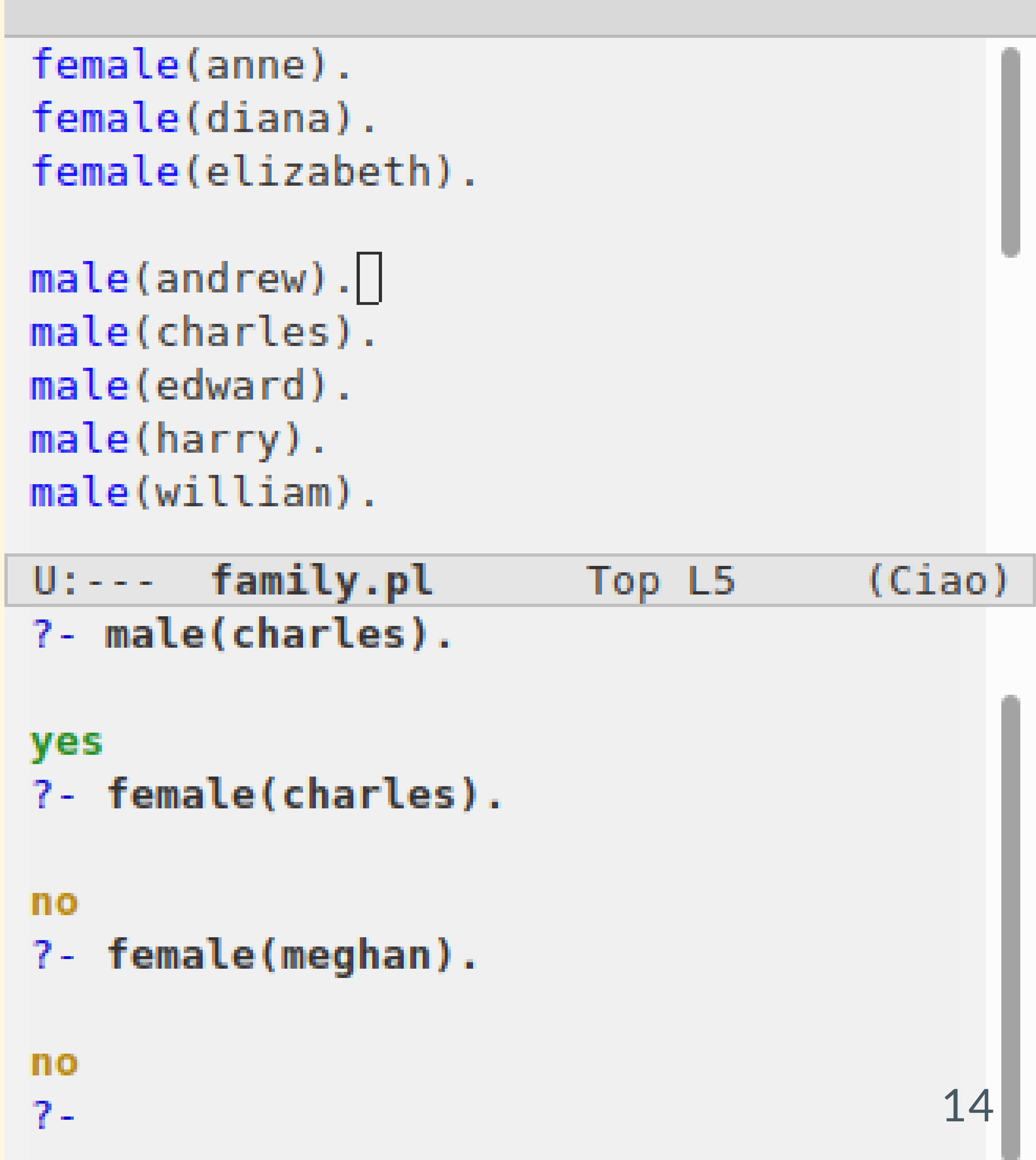
Let's consider the domain of genealogical relationships in the British Royal family, written as relations:

- facts: `female/1`, `male/1`, `parent/2`
- rules: `mother/2`, `father/2`

Closed world assumption

Anything that cannot be inferred with given data is assumed **false**.

Here this is illustrated for the query `?- female(meghan).`



```
female(anne).
female(diana).
female(elizabeth).

male(andrew).
male(charles).
male(edward).
male(harry).
male(william).

U:---  family.pl       Top L5       (Ciao)
?- male(charles).

yes
?- female(charles).

no
?- female(meghan).

no
?-
```

Computation mechanisms

Queries are answered by:

- **matching** (sub)goals against facts or rules
- **unifying** free **variables** with terms (NOT assignment)
- **backtracking** when (sub)goal matching fails

Let's briefly see these three mechanisms on the family tree example.

Example goal resolution

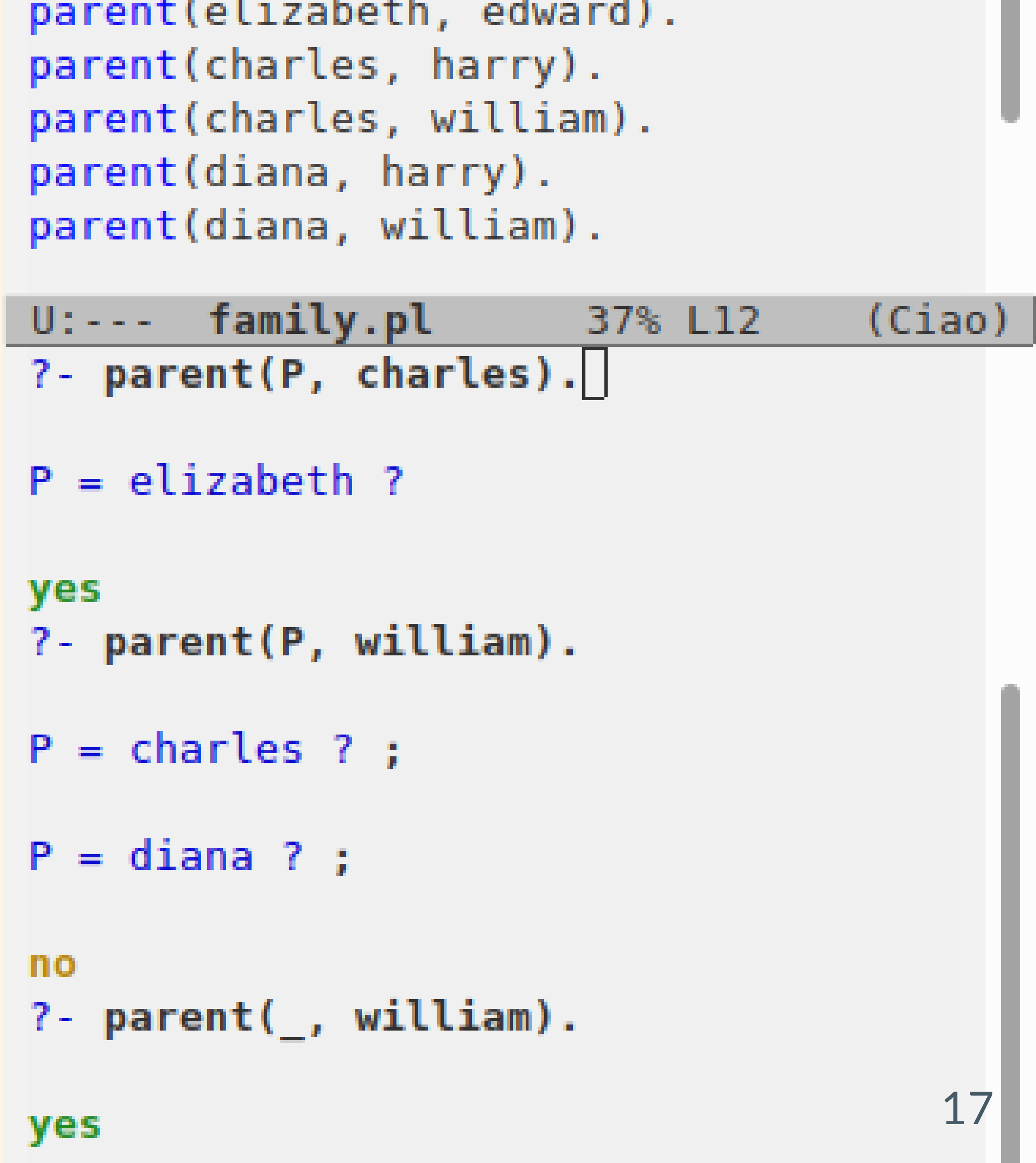
```
female(diana).                                  parent(diana, william).   % r1
mother(M,C) :- parent(M, C), female(M).         parent(diana, harry).     % r2
```

Answers to a query (initial goal) `?- mother(diana, C)` are computed by expanding sub-goals in a **search tree**:

```
                      mother(diana, C) {M=diana}
                              |
                      parent(diana,C),female(diana) {M=diana}
                      / (r1)          \ (r2)
parent(diana,william),female(diana)     parent(diana,harry),female(diana)
      {M=diana, C=william}                            {M=diana, C=harry}
```

Query to explore values

- to accept variable unification press ENTER
- to explore unification options press ; for backtracking
- use anonymous variables _ when you do not care about some values



```
parent(elizabeth, edward).
parent(charles, harry).
parent(charles, william).
parent(diana, harry).
parent(diana, william).

U:---  family.pl      37% L12     (Ciao)
?- parent(P, charles).

P = elizabeth ?


yes
?- parent(P, william).

P = charles ? ;

P = diana ? ;


no
?- parent(_, william).


yes
```

Unification and comparison

In Prolog there is no assignment, but terms can be unified and compared using:

- `=/2` - a binary unification operator
- `==/2` and `\==/2` - binary term comparison operators

Arithmetic operations are treated in a special way, stay tuned.

Unification (1/2)

Unification is instantiating variables by pattern matching:

- a constant only unifies with itself:

```
?- diana = diana.
yes
?- charles = diana.
no
```

- a free variable unifies with anything:

```
?- parent(elisabeth, charles) = Y.
Y = parent(elisabeth, charles) ?
yes
```

Unification (2/2)

Unification is instantiating variables by pattern matching:

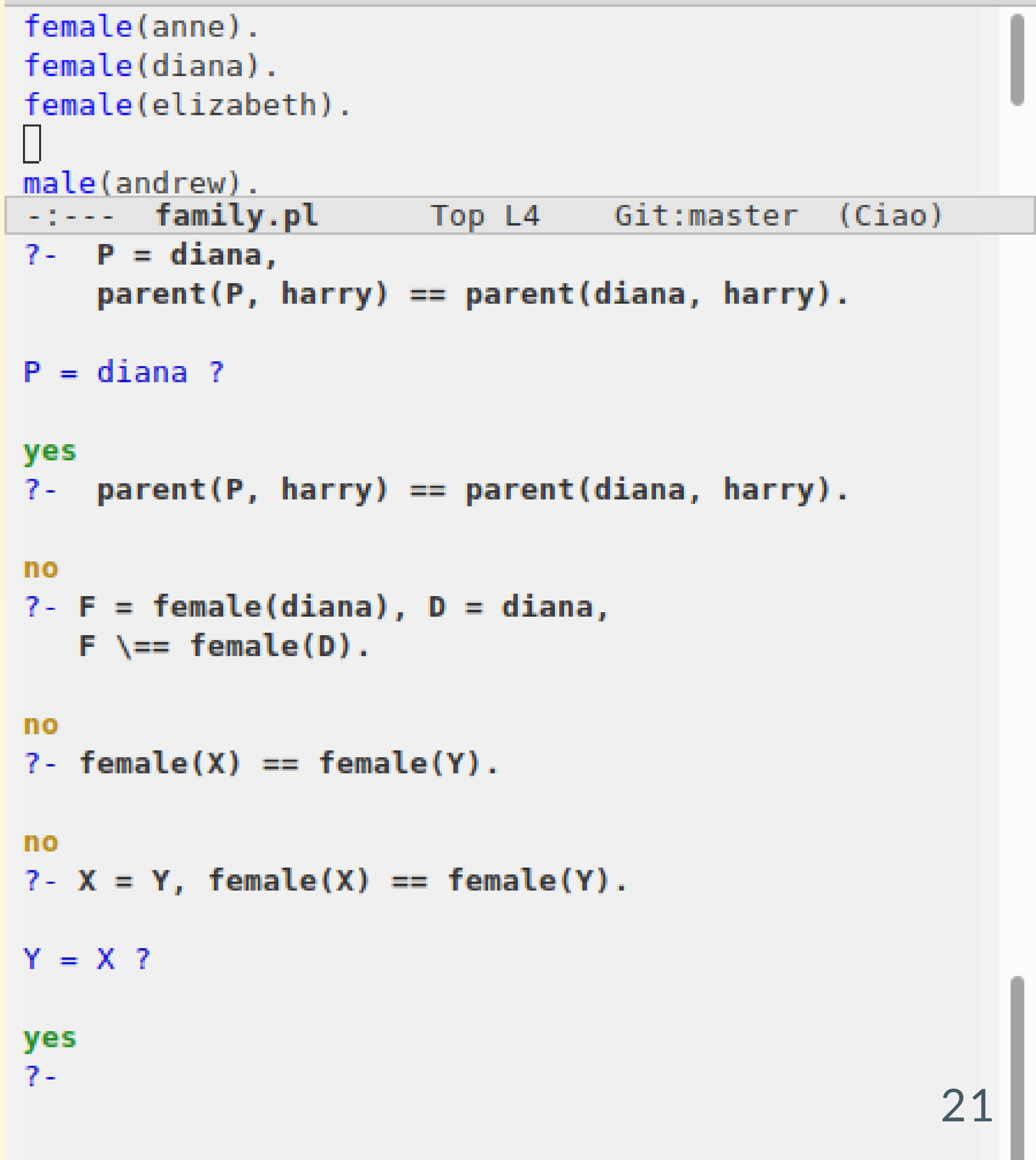
- a term unifies with another term only if it has the same functor name and number of arguments, and the arguments can be unified recursively:

```
?- parent(elisabeth, X) = parent(Y,Z).
Y = elisabeth,
Z = X ?
yes
```

Comparison

Comparison operators check if two terms are **strictly identical** (or not):

- functors match
- number of arguments for compound terms matches
- free variables are shared (same name in scope or explicitly unified)



```
female(anne).
female(diana).
female(elizabeth).

male(andrew).
-:---  family.pl       Top L4     Git:master  (Ciao)
?-  P = diana,
    parent(P, harry) == parent(diana, harry).

P = diana ?

yes
?-  parent(P, harry) == parent(diana, harry).

no
?- F = female(diana), D = diana,
   F \== female(D).

no
?- female(X) == female(Y).

no
?- X = Y, female(X) == female(Y).

Y = X ?

yes
?-
```

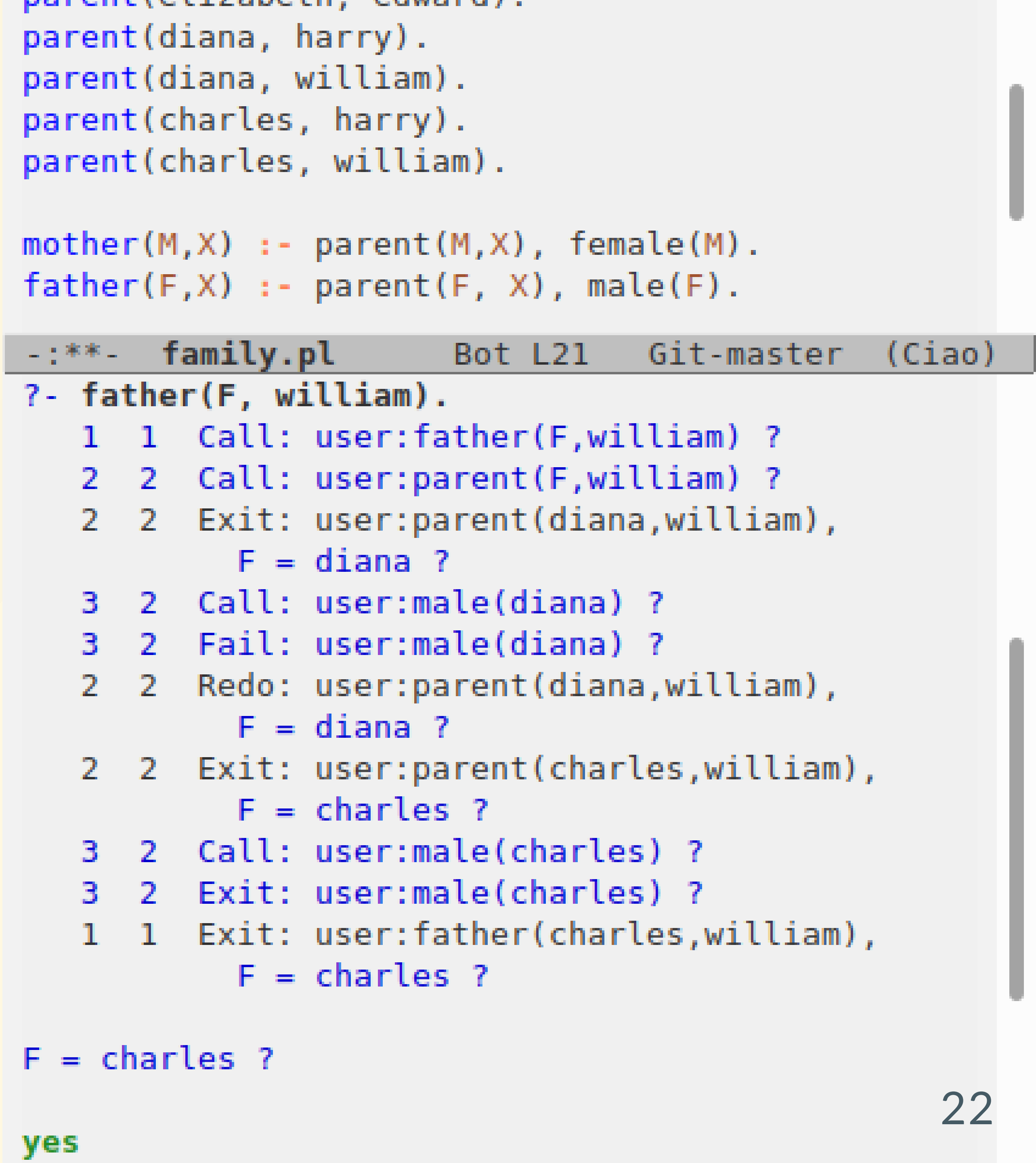
Backtracking

Prolog applies resolution in deterministic manner:

- (sub)goals are substituted left to right
- clauses are tried top-to-bottom

Consider the query:

`?- father(F, william)`



```
parent(diana, harry).
parent(diana, william).
parent(charles, harry).
parent(charles, william).

mother(M,X) :- parent(M,X), female(M).
father(F,X) :- parent(F, X), male(F).

-:**-  family.pl      Bot L21    Git-master   (Ciao)
?- father(F, william).
   1  1  Call: user:father(F,william) ?
   2  2  Call: user:parent(F,william) ?
   2  2  Exit: user:parent(diana,william),
            F = diana ?
   3  2  Call: user:male(diana) ?
   3  2  Fail: user:male(diana) ?
   2  2  Redo: user:parent(diana,william),
            F = diana ?
   2  2  Exit: user:parent(charles,william),
            F = charles ?
   3  2  Call: user:male(charles) ?
   3  2  Exit: user:male(charles) ?
   1  1  Exit: user:father(charles,william),
            F = charles ?

F = charles ?

yes
```

Disjunction (1/2)

Logical OR operator `;` can be used directly in the rules:

```
is_parent(P, C) :- mother(P, C).
is_parent(P, C) :- father(P, C).
```

can be written for convenience as:

```
is_parent(P, C) :- mother(P, C) ; father(P, C).
```

Disjunction (1/2)

```
female(diana).      parent(diana, harry).        mother(diana, harry).
male(charles).      parent(charles, harry).      father(charles, harry).

is_mother(M, C) :- parent(M, C), female(M).

is_parent(M, C) :- mother(M, C) ; father(M,C).
```

When designing the relations about domain objects think first:

- which way is it easier to *express* facts?
- which way makes it faster to *evaluate* queries?

Recursion

Recursive relations contain the same term in the head and the body:

```
ancestor(A, D) :- parent(A, D).                    % C1: base case
ancestor(A, D) :- parent(P, D), ancestor(A, P). % C2: recursive case
```

Remember the left-to-right, top-to-bottom literal and clause evaluation order?

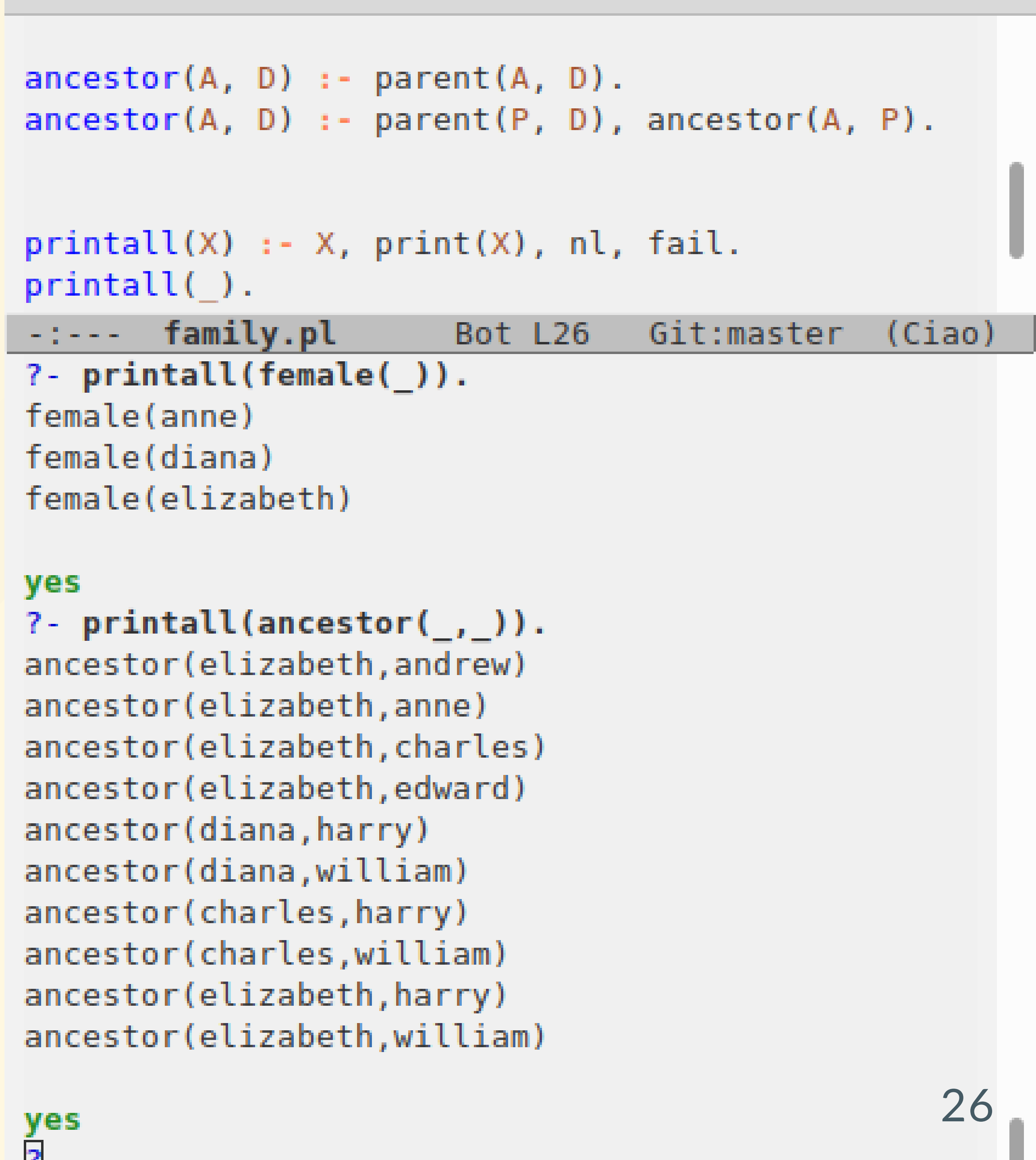
- be sure to write base case first
- and the recursive goal - last

Failure

Search can be controlled explicitly with `fail/0` special goal.

In the first clause of the predicate `printall/1` failture forces exploration of all possible variable unifications.

Note the I/O system predicates: `print/1`, `nl/0`



```
ancestor(A, D) :- parent(A, D).
ancestor(A, D) :- parent(P, D), ancestor(A, P).


printall(X) :- X, print(X), nl, fail.
printall(_).
-:---  family.pl      Bot L26    Git:master   (Ciao)
?- printall(female(_)).
female(anne)
female(diana)
female(elizabeth)

yes
?- printall(ancestor(_,_)).
ancestor(elizabeth,andrew)
ancestor(elizabeth,anne)
ancestor(elizabeth,charles)
ancestor(elizabeth,edward)
ancestor(diana,harry)
ancestor(diana,william)
ancestor(charles,harry)
ancestor(charles,william)
ancestor(elizabeth,harry)
ancestor(elizabeth,william)

yes
?
```

Cuts

Search can be also explicitly controlled by the **cut** operator `!/0`, that *commits* Prolog to a specific search path and controlls backtracking:

```
parent(C,P) :- mother(C,P), !. % mother is a valid parent
parent(C,P) :- father(C,P).
```

The `!` operator *prunes* the search tree by telling Prolog to discard:

- clauses below the clause in which the `!` appears
- all alternative solutions to the goals to the left of the `!`

Red and Green Cuts

- A **green cut** does not change the semantics of the program. It just eliminates useless searching:

```
max(X, Y, X) :- X > Y, !. % mutually exclusive cases, optmization
max(X, Y, Y) :- X =< Y.
```

- A **red cut** changes the semantics of your program. If you remove the cut, you can get incorrect results: `?- max(5,2,2).`

```
max(X, Y, X) :- X > Y, !. % not really mutually exclusive:
max(_, Y, Y).  % this clause will succeed if comarison before cut fails
```

Negation as failure

By default there is no negation: remember the closed world assumption?

Negation can be implemented by a combination of a (red) cut and fail:

```
not(X) :- X, !, fail. % if X succeeds, we fail
not(_).               % if X fails, we succeed
```

If the `!` is removed, the first clause will keep failing both when `X` succeeds and fails itself, and thus `X` will trivially succeed always.

Unification and comparison …and arithmetics

Apart from usual distinction between unification and comparison arithmetic operations are treated separately:

- `is/2` is a **function** from its second operand to its first
- `=:=/2` and `=\=/2` are arithmetic expression comparison operators

Valid goals: `X is 3 + 4`, `7 + 4 =\= 10 + 2`, `2 + 2 =:= 3 + 1`

Functions

User-defined functions are written in a relational style:

```
fact(0, 1).
fact(N, F) :- N > 0,
              N1 is N - 1,
              fact(N1,F1),
              F is N * F1.
```

Lists

Prolog lists can be written using 3 syntax flavors:

```
% FORMAL                    CONS PAIR            ELEMENT
.(a,[])                     [a|[]]               [a]
.(a,.(b,[]))                [a|[b|[]]]           [a,b]
.(a,.(b,.(c,[])))           [a|[b|[c|[]]]]       [a,b,c]
.(a,b)                      [a|b]                [a|b]
.(a,.(b,c))                 [a|[b|c]]            [a,b|c]
```

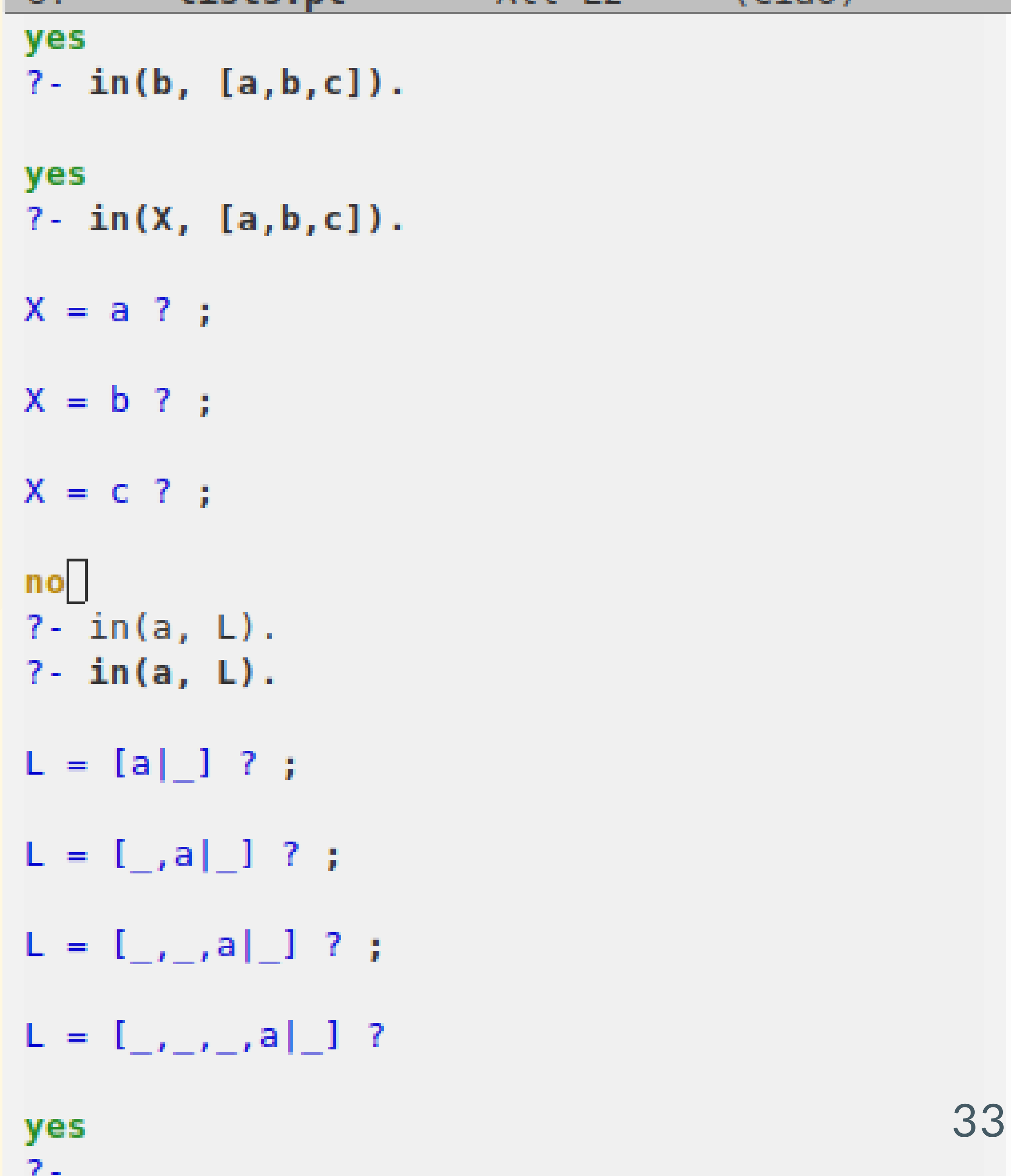
Lists consist of either an *empty list* `[]`, or a non-empty list (*e.g.* `[a | [b,c]]`) consisting of the head element (`a`) and the tail of the list (`[b,c]`). Free variables are allowed.

Pattern Matching in Lists

Consider predicate for checking list membership

```
in(X, [X | _ ]).
in(X, [ _ | L]) :- in(X, L).
```

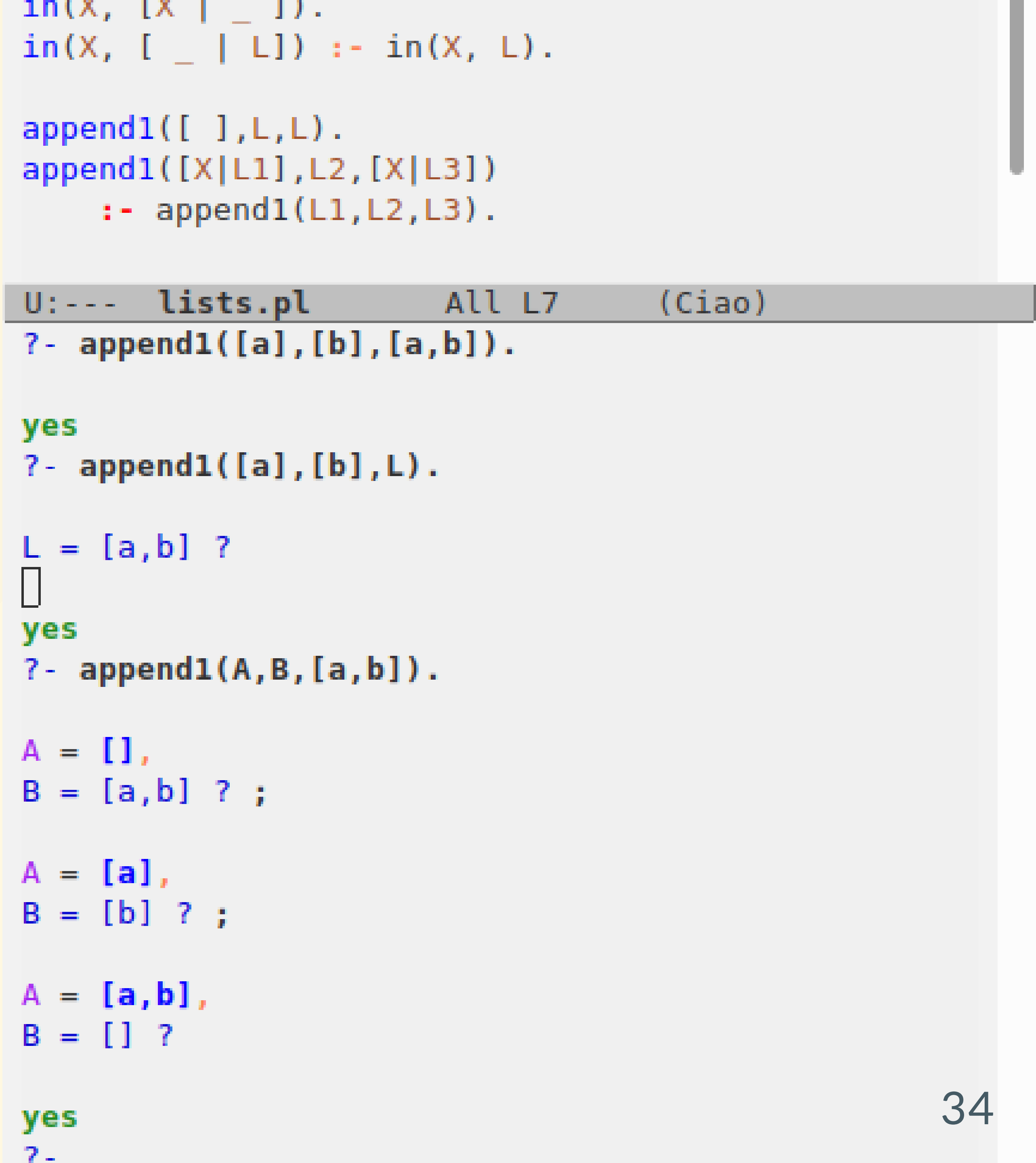
and several different queries:



```
yes
?- in(b, [a,b,c]).

yes
?- in(X, [a,b,c]).

X = a ? ;

X = b ? ;

X = c ? ;

no
?- in(a, L).
?- in(a, L).

L = [a|_] ? ;

L = [_,a|_] ? ;

L = [_,_,a|_] ? ;

L = [_,_,_,a|_] ?

yes
?-
```

No Order

Since there is no notion of input and return arguments, any argument can be both:



```
in(X, [X | _ ]).
in(X, [ _ | L]) :- in(X, L).

append1([ ],L,L).
append1([X|L1],L2,[X|L3])
     :- append1(L1,L2,L3).

U:---  lists.pl        All L7      (Ciao)
?- append1([a],[b],[a,b]).

yes
?- append1([a],[b],L).

L = [a,b] ?

yes
?- append1(A,B,[a,b]).

A = [],
B = [a,b] ? ;

A = [a],
B = [b] ? ;

A = [a,b],
B = [] ?

yes
?-
```

Resources

On logic programming:

- Chapter 12 of **Programming Languages** book by Kenneth C. Louden

On Prolog specifically:

- **The Art of Prolog** book by Leon S. Sterling and Ehud Y. Shapiro

On Ciao Prolog:

- **Ciao documentation page**