UNIVERSITÄT
BERN

11. Logic Programming Applications

Nataliia Stulova

-

- [ ] Efficiency in computations 👈
- [ ] Datalog
- [ ] CLP and numeric computaitons
- [ ] Assertions and program verification
- [ ] Natural language and parsing with DCGs

Green and red cuts revised: optimize the search

Geen cuts (discarding solutions we do not need)

E.g., personnel management software: one address is enough

```
address(X,Add) :- home_address(X,Add),!.
address(X,Add) :- business_address(X,Add).
```

- pay attention to variable unifications
- pay attention to declarative semantics
- solutions with and without green cuts should match

Green and red cuts revised: optimize the search

Red cuts (manipulating the search in a wrong way, **avoid**)

E.g., for a given year return a number of days (but *forgot to account for unification in the head*)

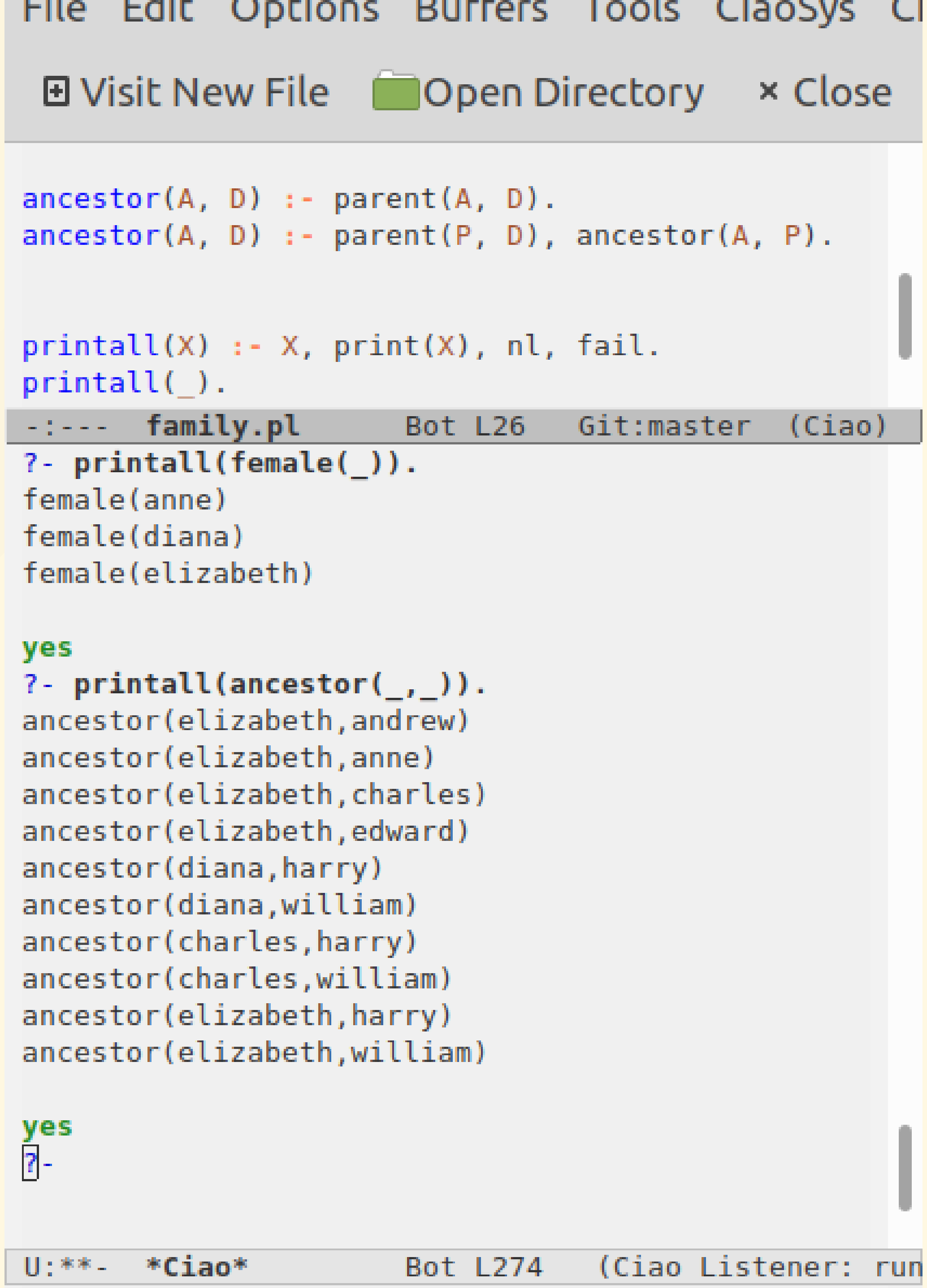
```
leap_year(Y) :- number(Y), 0 is Y mod 4.

days_in_year(Y,366) :- leap_year(Y),!.
days_in_year(_,365).                    % return 365 for any term?
```

queries that will succeed: `?- days_in_year(4, 365).` `?- days_in_year(a, D)`

Think sets, use lists

A lot of Prolog computations is producing sets of possible solutions to a query/goal, like with `printall/1` ➡️

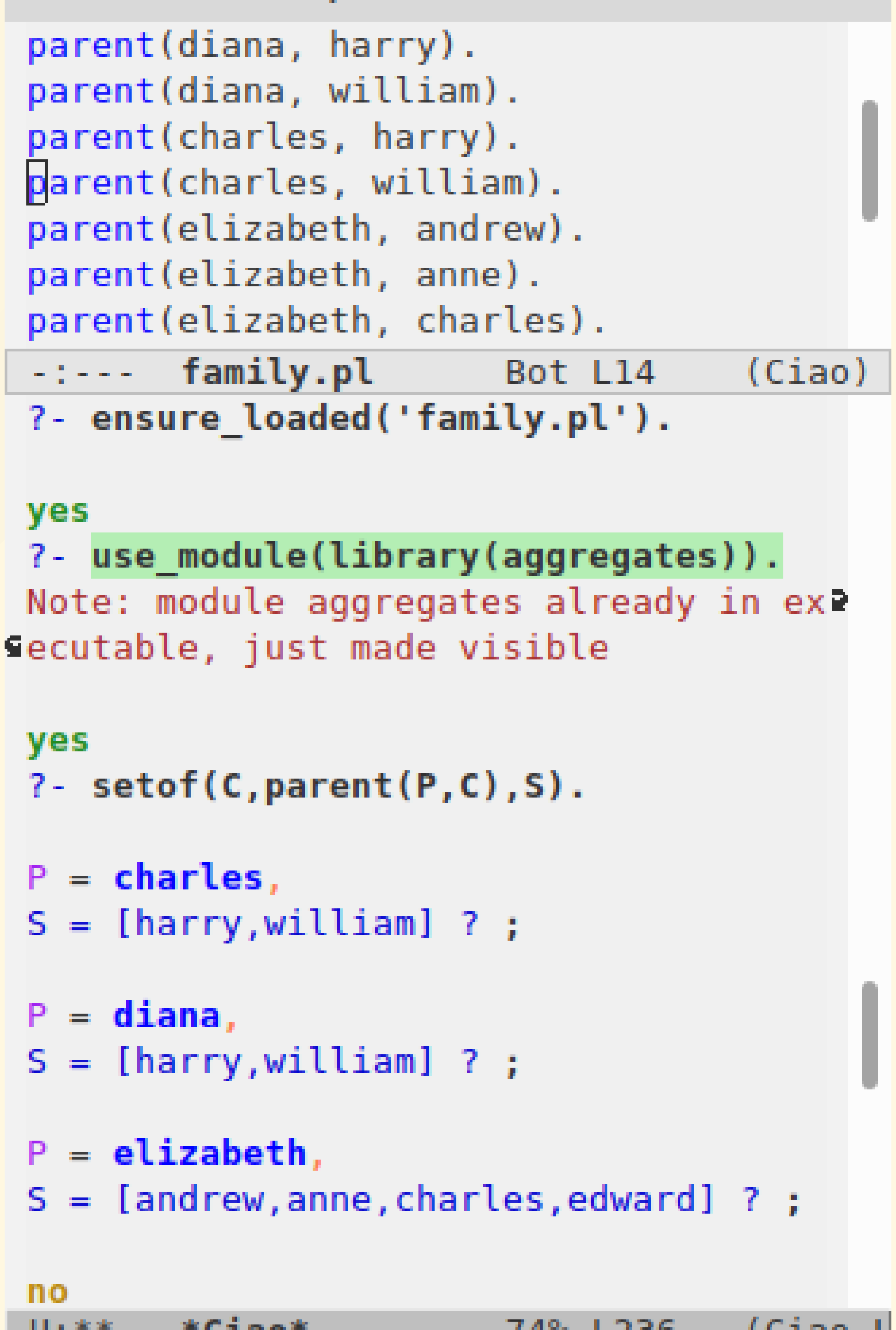
Lists are the native data structure that most intuitively represents sets.



Lists and aggregates

A number of system predicates is available to return sets of answers collected with backtracking:

- `setof/3` (and `bagof/3`) ➡️

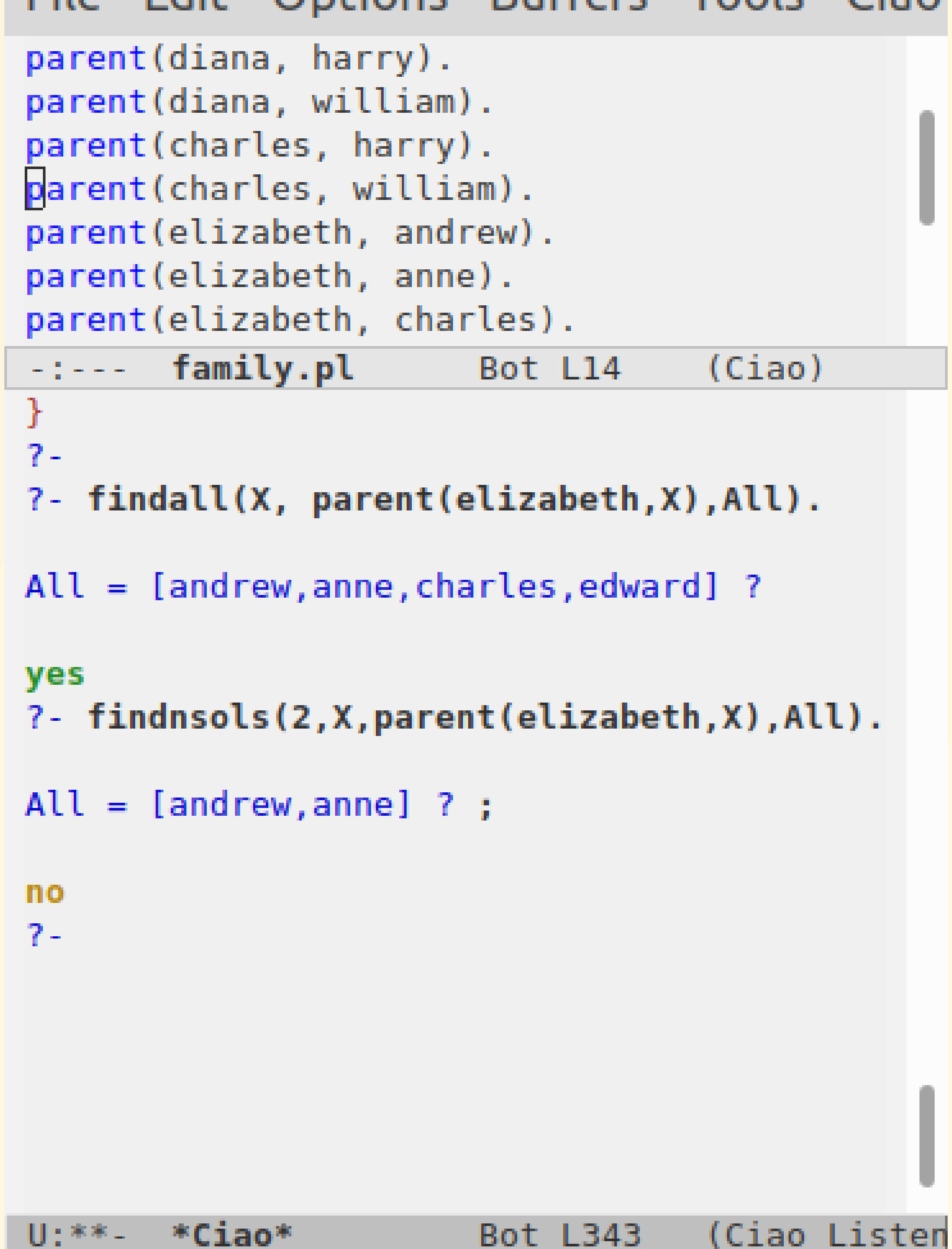


Lists and aggregates

A number of system predicates is available to return sets of answers collected with backtracking:

- `setof/3` (and `bagof/3`)
- `findall/3`
- `findnsols/4`
-

Ciao Aggregates

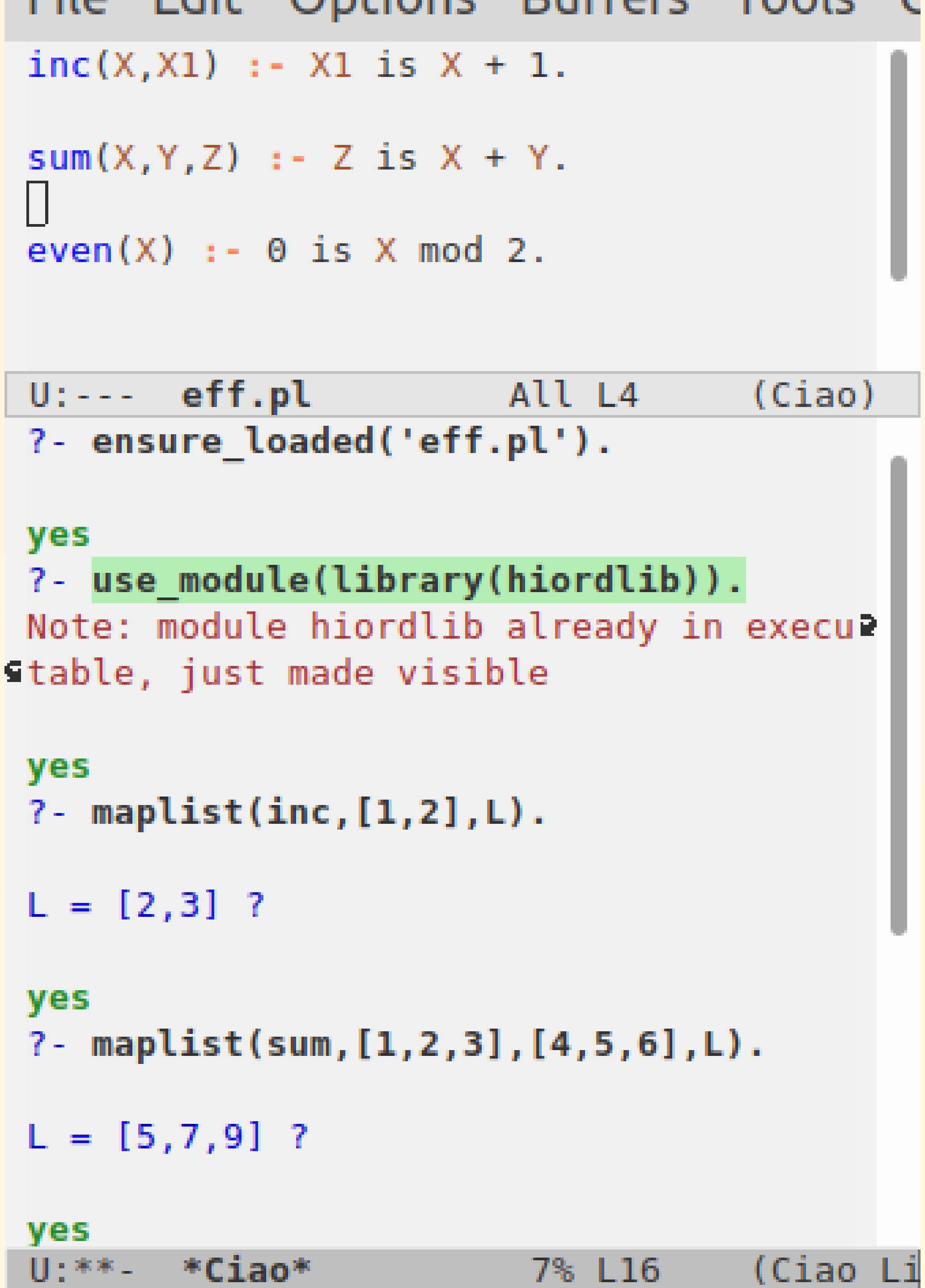


Higher-order

Processing *parallel* lists is very common, especially when using higher-order predicates like `maplist/N` ➡️

HO predicates accept other predicates (`inc/2`, `sum/3`) as arguments natively in Prolog, just make them visible in the scope (interpreter, module).

🔗[Ciao HO predicates]()



Higher-order

Processing *parallel* lists is very common, especially when using higher-order predicates like `filter/3`, `partition/4` ➡

```
inc(X,X1) :- X1 is X + 1.

sum(X,Y,Z) :- Z is X + Y.

even(X) :- 0 is X mod 2.
```

```
U:---  eff.pl        All L4      (Ciao)
?- filter(even,[1,2,3,4,5],L).

L = [2,4] ?

yes
?- partition(even,[1,2,3,4,5],L,R).

L = [2,4],
R = [1,3,5] ?

yes
?-
U:**-  *Ciao*        Bot L21     (Ciao Li
```

- ✅ Efficiency in computations
- ⬜ Datalog 👈
- ⬜ CLP and numeric computaitons
- ⬜ Assertions and program verification
- ⬜ Natural language and parsing with DCGs

Logic Programming + Relational Databases = Deductive Databases

Origins: the 1977 Symposium on Logic and Data Bases

Deductive databases have the advantage of making inference (*deduction*) of additional facts based on relations (facts and rules) already present in the database.

- data representation: relations (based on Horn clauses)
- query language: Datalog

Datalog VS Prolog

- recursion is allowed
- negation is allowed, but *only for facts*
- clause order does not matter
- not cut `!/0` operator to control search
- function symbols not allowed - cannot construct complex terms (*e.g.* `person(name(elisabeth), age(inf), ...)` has to be expressed as `person(elisabeth, inf, ...)`)
- queries are made on finite sets of values, so termination is guaranteed

Datalog queries and database operations

consider relations `P(x,y)`, `Q(x,y,z)`:

- intersection `I(x,y) :- P(x,y), Q(x,y,_).` (logical AND)
- union `U(x,y) :- P(x,y). ; U(x,y) :- Q(x,y,_).` (logical OR)
- difference `D(x,y) :- P(x,y), not Q(x,y,_).`
- projection `Px(x) :- P(x,_).`
- selection `S(x,y) :- Q(x,y,_), x > 10.`
- product `PR(x,y,z,v,w) :- P(x,y), Q(z,v,w).`
- join `J(x,y,z) :- P(x,y), Q(y,z,_).`
- …

Datalog systems - few examples

LogicBlox

🔗 logicblox.com - a commercial implementation of Datalog used for web-based retail planning and optimization

Datomic
Data, meet Simple

🔗 datomic.com - a transactional database with a flexible data model, elastic scaling, and rich queries

... and many more systems with Datalog components

- ✅ Efficiency in computations
- ✅ Datalog
- ⬜ CLP and numeric computaitons 👈
- ⬜ Assertions and program verification
- ⬜ Natural language and parsing with DCGs

Constraint satisfaction problems

In the fields of *artificial intelligence* and *operations research* there is a need in answering questions in different domains that specify a number of **constraints** for an answer:

- route planning with time or price budget
- diet meal preparation accounting for calories intake
- solving chess problems (*e.g.* N-queens)
- map coloring

Constraint satisfaction problems are typically solved using a form of **search**.

Constraint Logic Programming (CLP)

CLP is an extension of logic programming that includes **constraint satisfaction** in the computations:

CLP = Prolog + Solver(for a given domain)

Some domains:

- finite (CLPFD) - *e.g.* the familty tree
- rational numbers (CLP(Q))
- real numbers (CLP(R))

CLP(R) in Ciao 1/2

As a language extension CLP functionality is available as a library that defines special operators: `.=./2` (equals), `.<./2` (less than), *etc.*

Example: vector dot product ➡️ $(x_1, \ldots, x_N) \cdot (y_1, \ldots, y_N) = x_1 \cdot y_1 + \ldots + x_N \cdot y_N$

```
:- use_package(clpr).

prod([],[], Result) :-
    Result .=. 0.
prod([X|Xs],[Y|Ys], Result) :-
    Result .=. X * Y + Rest,
    prod(Xs, Ys, Rest).
```

```
U:---  clp.pl          All L7       (Ciao)
Ciao 1.20.0 [LINUXx86_64]
?- ensure_loaded('clp.pl').

yes
?- prod([2,3], [4,5], P).

P.=.23.0 ?

yes
?- prod([2,7,3],[Vx,Vy,Vz],0).

Vz.=. -0.6666666666666666*Vx-2.33333
3333333333*Vy ? ;

no
```

CLP(R) in Ciao 2/2

Another example: solving systems of linear equations

$$3x + y = 5$$
$$x + 8y = 3$$

To solve this system we reuse the dot product relation for each equation ➡️

🔗[Ciao Language Extensions]()

```
:- use_package(clpr).

prod([],[], Result) :-
    Result .=. 0.
prod([X|Xs],[Y|Ys], Result) :-
    Result .=. X * Y + Rest,
    prod(Xs, Ys, Rest).
```

```
U:---  clp.pl          All L7      (Ciao)
?- prod([2,7,3],[Vx,Vy,Vz],0).

Vz.=. -0.6666666666666666*Vx-2.33333
3333333333*Vy ? ;

no
?- prod([3,1],[X,Y],5),
   prod([1,8],[X,Y],3).

Y.=.0.17391304347826075,
X.=.1.608695652173913 ?

yes
```

- ✅ Efficiency in computations
- ✅ Datalog
- ✅ CLP and numeric computaitons
- ⬜ Assertions and program verification 👈
- ⬜ Natural language and parsing with DCGs

Program correctness: testing and verification

Two (complementary) approaches to checking correctness of program behavior

Testing

- at run time
- for specific inputs (*e.g.* `min(-2,5,-2)`)

Verification

- at *compile* or run time (or both)
- for classes of inputs (*e.g.*, `min(+,-,-)`)

Software verification

- define **properties** in a domain of interest: memory addresses, numeric ranges of array indices, dangling pointers, energy consumption constratins...
- write program **specifications** using these properties (often in some formal laguage)
- **check** the specifications with some technique: code instrumentation, theorem proving, logical inference

Specification examples

C

```c
int magic ( int size , char *format )
    assert ( size <= LIMIT ) ;
```

Racket

```racket
( define / contract ( our-div num denom )
( number ? ( and / c number ? ( not / c zero ?) ) . -> . number ?)
(/ num denom ) )
```

Prolog

```prolog
:- pred append(A,B,C) : list(A), list(B), var(C)
                      => size(ub,C,length(B)+length(A))
                       + cost(ub,steps,length(A)+1).
```

Horn clause-based program verification

- programming language and its specification are based on same formal representation
- nowadays a number of mature analysis techniques and tools exist for logic programs analysis and verification
- for several high-level languages (C/C++, Java) their *intermediate representation* (produced by the compiler) can be straightforwardly translated to Horn clauses

Example: factorial (1/2)

Consider the factorial function in XC, a dialect of C for microcontroller programming:

```
#pragma check fact(n): (1 <= n) ==> (6.0 <= energy_nJ  <= 2.3*n+9.0)

int  fact(int N) {
    if (N <= 0)  return 1;
    return N * fact(N - 1);
}
```

Properties of interest: energy consumption estimates.

Example: factorial (2/2)

```
.
.
.
.

<fact>:
001:  entsp  0x2
002:  stw    r0,  sp[0x1]
003:  ldw    r1,  sp[0x1]
004:  ldc    r0,  0x0
005:  lss    r0,  r0,  r1
006:  bf     r0,  <008>
```

```
:- check pred fact(N, Ret)
   : intervals(nat(N),[i(1,inf)])
   + costb(energy_nJ,6.0,
           2.3*nat(N)+9.0).

fact(R0,R0_3) :-
   entsp(0x2),
   stw(R0,Sp0x1),
   ldw(R1,Sp0x1),
   ldc(R0_1,0x0),
   lss(R0_2,R0_1,R1),
   bf(R0_2,0x8),
   fact_aux(R0_2,Sp0x1,R0_3,R1_1).
```

ISA (instruction set architecture) instructions for the XC program and respective Horn clause representation with a specification

Some HC-based verification tools

for Java, XC, C and C++ the Ciao preprocessor - CiaoPP - offers *abstract interpretation*-based analyses over several domains: types, variable instantiation, bounds on computational and energy costs *etc*

JayHorn is a software *model checking* tool for Java that tries to find a proof that certain bad states in a Java program are never reachable.

- ✅ Efficiency in computations
- ✅ Datalog
- ✅ CLP and numeric computaitons
- ✅ Assertions and program verification
- ⬜ Natural language and parsing with DCGs 👈

IBM Watson

```
POETS & POETRY: He was a bank clerk in the Yukon before he published
“Songs of a Sourdough” in 1907
```

"We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations"

```
lemma(1, "he").                       partOfSpeech(1,pronoun).    subject(2,1).
lemma(2, "publish").                  partOfSpeech(2,verb).       object(2,3).
lemma(3, "Songs of a Sourdough").     partOfSpeech(3,noun).       ...
```

🔗[Natural Language Processing With Prolog in the IBM Watson System]()

Languages and grammars (1/2)

Every language has a grammar - a set of elements and rules on combining those elements.

Consider English language and some of its elemets (parts of speech):

- articles: *definite* `the`, and *indefinite* `a` and `an`
- nouns: *proper* `alice`, and *common* `cat`, `fish`, `bat`
- pronouns: `she`, `whose`, `their`
- verbs: `play`, `eats`
- …

Languages and grammars (2/2)

We need to add some rules to combile language elements:

$$sentence \rightarrow noun\_phrase, verb\_phrase$$
$$noun\_phrase \rightarrow article, noun$$
$$verb\_phrase \rightarrow verb, noun\_phrase$$

Let's try to build sentences with the elements we have defined so far:

```
  a  cat  eats  a  bat             s  = sentence
\ \_np_/  \  \__np__/ / /          np = noun_phrase
 \         \___vp____/ /           vp = verb_phrase
  \_______s___________/
```

Sentences as lists

We can express sentences as lists of elements provided by Prolog facts and rules:

```
s(C)  :- np(A),  vp(B), append(A,B,C).
np(C) :- det(A), n(B),  append(A,B,C).
vp(C) :- v(A)  , np(B), append(A,B,C).

det([a]).      n([cat]).
det([the]).    n([fish]).    v([eats]).
```

Why lists? Sentences can be of arbitrary length and designing terms for each possible structure is not feasible.

Grammar in Prolog v1

We can both parse and generate sentences with this implementation ➡️

However, this is a computation-heavy implemetnation.

Alternative specialized representation: **difference lists**

```
File  Edit  Options  Buffers  Tools  Ciao
s(C)  :- np(A),  vp(B), append(A,B,C).
np(C) :- det(A), n(B),  append(A,B,C).
vp(C) :- v(A)  , np(B), append(A,B,C).

det([a]).      n([cat]).
det([the]).    n([fish]).    v([eats]).

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

```
?- s(S).

S = [a,cat,eats,a,cat] ? ;

S = [a,cat,eats,a,fish] ?

yes
?- s([a,fish,eats,a,fish]).

yes
?- s([a,bat,eats,a,fish]).

no
?-

U:**-  *Ciao*          Bot L191    (Ciao Liste
```

Difference lists

Prolog's special way of representing lists for language parsing and generation tasks:

- `X-X` is the empty list `[]`
- `[a,b,c]-[]` is the list `[a,b,c]`
- `[a,b,c,d]-[d]` is the list `[a,b,c]`
- `[a,b,c|T]-[T]` is the list `[a,b,c]` - with a free tail

Think of it as a literal difference between the first and the second list.

Definite clause grammars

In addition to difference lists, Prolog has a special notation for grammar representation, that implicitly uses difference lists:

```
s --> np, vp.
```

is an expansion of a difference lists version:

`s(A-C) :- np(A-B), vp(B-C).` (or `s(S-[]) :- np(S-VP), vp(VP-[]).`)

which is in turn a from of:

```
s(S) :- np(NP), vp(VP), append(NP,VP,S).
```

Grammar in Prolog v2

Notice how we still need to provide the two list arguments in the query ➡️ `?-s([a,cat,eats,a,fish],[]).`

File Edit Options Buffers Tools Ciao

```
:- use_package(dcg).

s --> np, vp.
np --> det, n.
vp --> v, np.

det --> [the].  det --> [a].
n   --> [cat].  n   --> [fish].
v   --> [eats].
```

U:--- dcg2.pl All L6 (Ciao)

```
?- ensure_loaded('dcg2.pl').

yes
?- s([a,cat,eats,a,fish],[]).

yes
?- s([a,cat|Y],[]).

Y = [eats,the,cat] ? ;

Y = [eats,the,fish] ?

yes
?-
```

U:**- *Ciao* Bot L328 (Ciao Liste

- ✅ Efficiency in computations
- ✅ Datalog
- ✅ CLP and numeric computaitons
- ✅ Assertions and program verification
- ✅ Natural language and parsing with DCGs