Capitolo 3
Il paradigma imperativo vs il paradigma dichiarativo

Corso di Laurea Magistrale in Ingegneria Informatica

Prof. ENRICO DENTI

Dipartimento di Informatica – Scienza e Ingegneria (DISI)

IL PARADIGMA IMPERATIVO

- Nel paradigma imperativo, un programma è una *sequenza di ordini* (dal latino *imperare)*
  - intuitivo per noi (comandare ci viene facile..)
  - immediato da mappare su una macchina (che "esegue" mini-azioni)
  - MA un elenco di ordini è pensato per una ben precisa situazione: è intrinsecamente *non invertibile*
    - per esprimere la situazione "inversa", occorre un elenco di ordini completamente diverso, anche se in origine la relazione fra i dati era simmetrica!
- Esempio: l'equazione x = y-2
  - in matematica è simmetrica: dato y, ricavo x – o viceversa
  - ma un programma imperativo pensato per ricavare x non può essere usato "al contrario" per ricavare y: va rifatto ex novo!

IL PARADIGMA IMPERATIVO

- L'equazione x = y-2
  - il programma imperativo pensato per ricavare x

```
int xFromY(int y) {return y-2;}
```

  - il programma imperativo pensato per ricavare y

```
int yFromX(int x) {return x+2;}
```

  - entrambi i casi assieme

```
int solve(int x, int y){…}
```

come distinguere quale è l'input? → null o Optional..?

```
int solve(Integer x, Integer y){
  if (x==null && y!=null) return y-2;
  if (x!=null && y==null) return x+2;
  if (x==null && x==null) ???
}
```

- Necessario **sgranare** le situazioni
- **Controllo esplicito** su ogni caso

VERSO ALTRI PARADIGMI: IL PARADIGMA DICHIARATIVO

- Il paradigma imperativo non è l'unico possibile
- Nel paradigma *dichiarativo,* non si esprimono ordini: si esprimono le ***relazioni fra le entità***
  - un po' meno intuitivo per noi (all'inizio)
  - MA *intrinsecamente invertibile* perché non "battezza" fin dall'inizio quali elementi siano input e quali output
  - si limita ad *affermare ciò che è vero*
- Esempio: l'equazione x = y-2
  - in matematica è simmetrica: dato y, ricavo x – o viceversa
  - in un programma dichiarativo si può esprimere la relazione *as is*, demandando a runtime la scelta di usarla in un verso o nell'altro
  - si delega il controllo di dettaglio alla macchina

UN ALTRO ESEMPIO
APPEND DI DUE LISTE

- Se L1=[a,b,c], L2=[d,e], append(L1,L2)=[a,b,c,d,e]
- Imperativamente, occorre
  - battezzare a priori gli argomenti di input e output
  - oppure distinguere i tre casi
- ESEMPIO: il metodo **`addAll`** di Java Collection Framework:

```
public boolean addAll(int index, Collection<? extends E> c) {
```

USO: **`l3 = l1.addAll(l2);`**

ARGOMENTI

input: liste **`l1`** ed **`l2`**

output: lista **`l3`**

UN ALTRO ESEMPIO
APPEND DI DUE LISTE

```
public boolean addAll(int index, Collection<? extends E> c) {
    checkPositionIndex(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    if (numNew == 0)
        return false;

    Node<E> pred, succ;
    if (index == size) {
        succ = null;
        pred = last;
    } else {
        succ = node(index);
        pred = succ.prev;
    }

    for (Object o : a) {
        @SuppressWarnings("unchecked") E e = (E) o;
        Node<E> newNode = new Node<>(pred, e, null);
        if (pred == null)
            first = newNode;
        else
            pred.next = newNode;
        pred = newNode;
    }

    if (succ == null) {
        last = pred;
    } else {
        pred.next = succ;
        succ.prev = pred;
    }

    size += numNew;
    modCount++;
    return true;
}
```

Approccio imperativo: si esprime *IL CONTROLLO* per guidare la computazione, *istruzione per istruzione*

- Error prone
- *Difficile da seguire e debuggare*
- Ruolo degli argomenti prestabilito
  - alcuni sono input
  - altri sono output

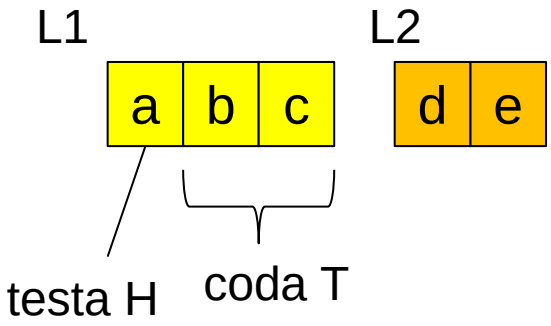
Approccio ispirato dal basso

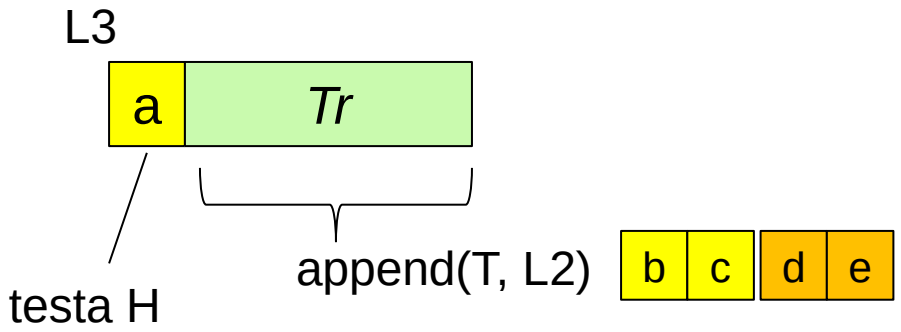
- macchina vista come *mero esecutore,* incapace di autonomia
- *ossessione del controllo*

odio di denti per i linguaggi di basso livello noted

UN ALTRO ESEMPIO
APPEND DI DUE LISTE

- Dichiarativamente,
  basta esprimere la relazione fra le liste coinvolte
  - [ ] è l'elemento neutro dell'append — append([ ],L) = L
  - in generale
    - se la lista L1 ha per testa H e per coda T, — L1 = [H | T]
      il risultato L3 ha per testa H… — L3 = [H | Tr]
      …. e per coda Tr l'append di T ed L2 — Tr = append(T,L2)





UN ALTRO ESEMPIO
APPEND DI DUE LISTE

- In Prolog ciò assume la forma di relazione fra tre elementi, espressa in *due regole:*

```
append([], L, L).
```

Fatto noto: appendendo la lista vuota a L, si ri-ottiene L.

```
append([H|T], L, [H|Tr]) :- append(T,L,Tr).
```

Regola (implicazione logica):

se la prima lista ha per testa H e per coda T	ossia L1 = [H | T]
e la seconda lista si chiama L	ossia L2 = L
il risultato è una lista avente testa H e coda Tr	ossia L3 = [H|Tr]
dove Tr è il risultato dell'append di T ed L	Tr = append(T,L)

ovvero:

`append([H|T], L, [H|Tr])` è vera SE (PURCHÉ)
sia vera `append(T,L,Tr)`

o anche, più compattamente:

`append(T,L,Tr)` *implica* `append([H|T],L,[H|Tr])`

UN ALTRO ESEMPIO
APPEND DI DUE LISTE

In Prolog, sono *due regole*:

```
append([],L,L).

append([H|T],L,[H|Tr]) :-
      append(T,L,Tr).
```

A parole:

- appendendo una lista L a una lista vuota [ ], si ri-ottiene sempre L
- appendendo L a una lista la cui testa sia H e la cui coda sia T, si ottiene una nuova lista avente per testa H e per coda il concatenamento (append) di T ed L.

Approccio dichiarativo:
si danno *le regole* per computare,
ma non si esplicita COME usarle.
SI DELEGA IL CONTROLLO.

- Per ogni caso, una regola
- *Semplice da seguire e debuggare*
- Ruolo degli argomenti *reversibile*
  - come in matematica
  - "input" sono gli argomenti noti, "output" quelli ignoti

Maggiore livello di astrazione

- la macchina gestisce il controllo
- *focus sull'obiettivo, non sul "come"*

UN ALTRO ESEMPIO
APPEND DI DUE LISTE

```
public boolean addAll(int index, Collection<? extends E> c) {
```

USO: `l3 = l1.addAll(l2);`

ARGOMENTI

input: liste `l1` ed `l2`

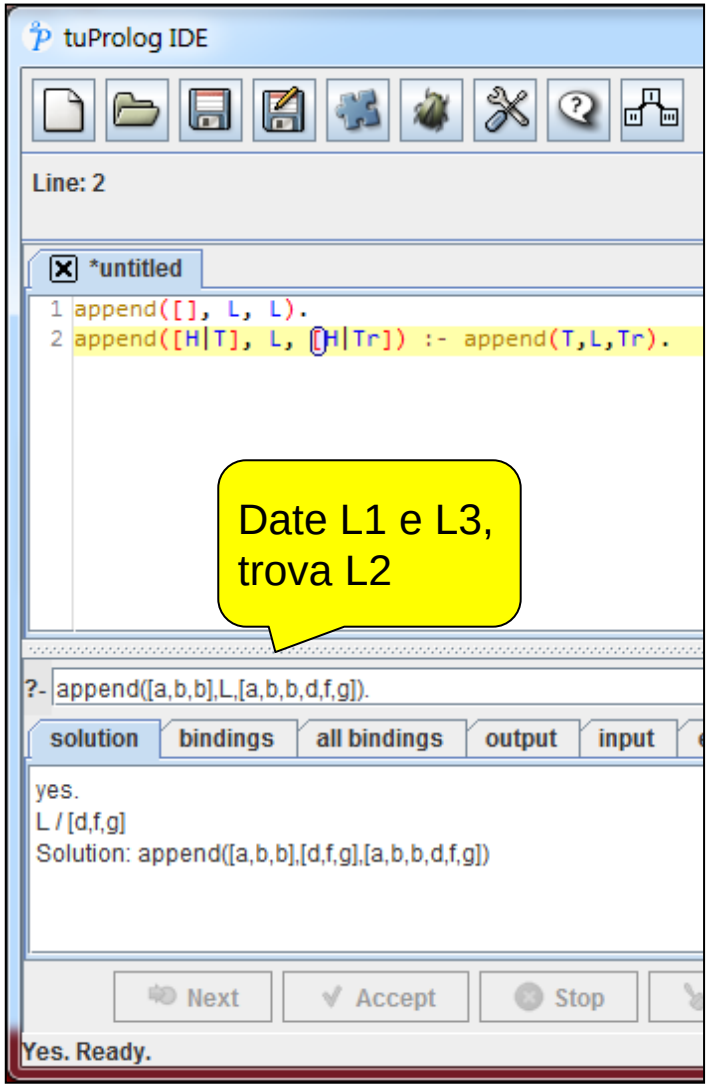
output: lista `l3`

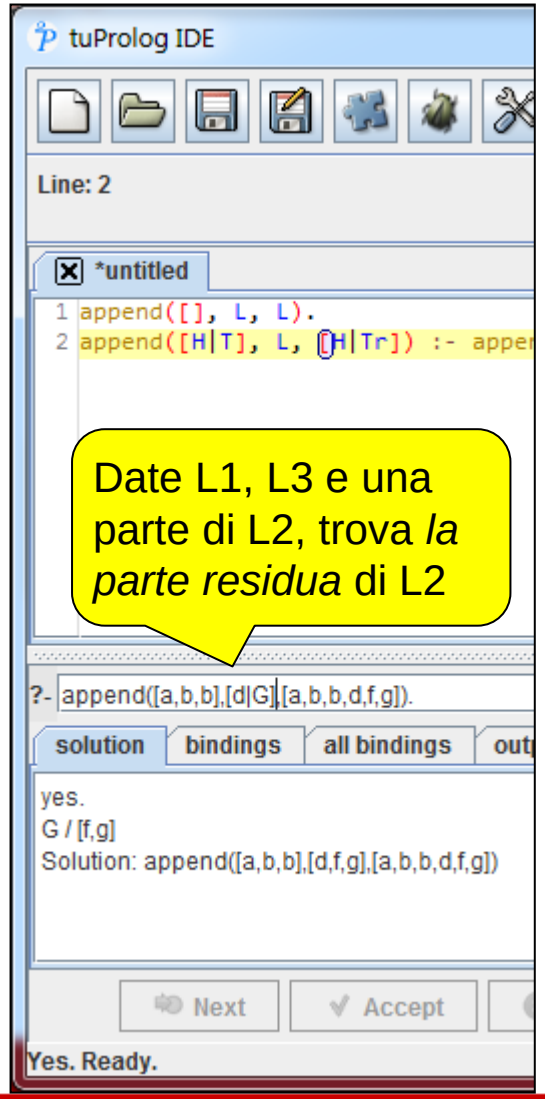
```
append([],L,L).
append([H|T],L,[H|Tr]) :-
        append(T,L,Tr).
```

USO: trovare i valori delle variabili *tali che*…

```
?- append([a,b], [c,4,d], R).
Solution: R / [a,b,c,4,d]
?- append([a,b], X, [a,b,c,d]).
Solution: X / [c,d]
```

UN ALTRO ESEMPIO
APPEND DI DUE LISTE





E se lo interrogassimo con tutte variabili..?

```
?- append(X,Y,Z).
```

Chissà… ☺

MINI-CORSO DI PROLOG IN 6 SLIDE (1)

- Un *programma Prolog* è un insieme di *regole (teoria logica)* espresse secondo la seguente notazione:

```
testa :- corpo.
```

- L'operatore `:-` esprime l'implicazione logica **(←)**: informalmente, *se il corpo è vero, allora anche la testa è vera* (non viceversa)
- Se il corpo manca, si considera *true* e quindi la testa è sempre vera
  In tal caso viene chiamata *fatto* e l'operatore `:-` si omette

```
fatto.
```

- La ***testa*** ha la forma ***funtore(lista_argomenti)***

```
p(a,12,X) :- corpo.
```

- La lista argomenti può mancare, nel qual caso le parentesi si omettono:

```
p :- corpo.
```

MINI-CORSO DI PROLOG IN 6 SLIDE (2)

- Il corpo è una *congiunzione* di termini separati da *virgole*
  Ogni termine ha anch'esso la forma ***funtore(lista_argomenti)***

```
p(a,12,X) :- q(a), r(13), s(X,1).
```

- Testa e corpo possono contenere *variabili* (iniziano con maiuscola, es. **X**)

```
p(a,12,X) :- q(a), r(13), s(X,1).

s(Y,X) :- q(X), r(Y).
```

- Lo scope di una variabile è la *singola regola*: la variabile **X** nella prima regola *non ha nulla a che fare* con la variabile **X** nella seconda regola.
- Se occorrono termini che iniziano per maiuscola, o che contengono caratteri non standard, occorre quotarli con apici:

```
p('Alfa',12, ':') :- q(X), q(Y).
```

MINI-CORSO DI PROLOG IN 6 SLIDE (3)

ESEMPIO 1

A chi piace cosa:

```
likes(mary, food).
likes(mary, wine).
likes(john, wine).
likes(john, mary).
```

Assiomi (fatti).

SEMANTICA INFORMALE

- **Y è padre di C** se
  Y è un uomo e inoltre
  Y è genitore di C
- **X è madre di C** se
  X è una donna e inoltre
  X è genitore di C

ESEMPIO 2

Figli e genitori:

Assiomi (fatti)

```
man(adam).
man(peter).
woman(mary).
woman(eve).
```

parent(genitore, figlio)

```
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).
```

Regole

```
father(Y,C):-
 man(Y), parent(Y,C).

mother(X,C):-
 woman(X),parent(X,C).
```

MINI-CORSO DI PROLOG IN 6 SLIDE (4)

- **Come un database**, il sistema Prolog ha una *base di conoscenza*
- **MA, a differenza di un database**, la conoscenza non comprende solo *fatti* ma anche *regole per dedurre nuova conoscenza*
- Quindi, le possibili query possono *non solo* recuperare tuple dal db, come:

```
?- woman(X).
```

ma anche recuperare tuple *da sintetizzare al volo.*

ESEMPIO 2

Figli e genitori:

```
man(adam).
man(peter).
woman(mary).
woman(eve).
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).

father(Y,C):-
 man(Y), parent(Y,C).

mother(X,C):-
 woman(X),parent(X,C).
```

Assiomi (fatti)

parent(genitore, figlio)

Regole

MINI-CORSO DI PROLOG IN 6 SLIDE (5)

QUERY

Interroghiamo il sistema:

```
?- father(X,paul).
X=adam
yes

?- father(eve,paul).
no
```

- 1ª query: X unifica con F, *paul* unifica con C e si propaga indietro *X=adam*
- 2ª query: *eve* unifica con F, *paul* unifica con C, ma il check *man(eve)* è falso →la query fallisce

ESEMPIO 2

Figli e genitori:

```
man(adam).
man(peter).
woman(mary).
woman(eve).
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).

father(Y,C):-
 man(Y), parent(Y,C).

mother(X,C):-
 woman(X),parent(X,C).
```

MINI-CORSO DI PROLOG IN 6 SLIDE (6)

QUERY

Interroghiamo il sistema:

```
?- father(adam,X).
X=peter ? ;
X=paul ? ;
no
```

- 1° tentativo: X unifica con C, *adam* unifica con F e si propaga indietro X=*peter* *ma non è l'unica soluzione*
- se viene accettata, ok; se si chiede una alternativa, *si disfa tutto e si rifà*
- 2° tentativo: il subgoal *parent(adam,C)* seleziona un figlio diverso → *X=paul*

ESEMPIO 2

Figli e genitori:

Esplorazione di soluzioni multiple

```
man(adam).
man(peter).
woman(mary).
woman(eve).
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).

father(F,C):-
 man(F), parent(F,C).

mother(M,C):-
 woman(M),parent(M,C).
```

E SI PUÒ CONTINUARE (CON ATTENZIONE…)

QUERY

Interroghiamo il sistema:

```
?- siblings(peter,paul).
yes
?- siblings(X,paul).
X / peter ? ;
X / paul ? ;
X / paul ? ;
no
```

OOPS! Regola errata: non abbiamo detto che U e V dovessero essere diversi!
Inoltre, poiché i genitori sono due, la risposta con **paul** è doppia ☹

ESEMPIO 3

Fratelli e sorelle

```
man(adam).
man(peter).
woman(mary).
woman(eve).
parent(adam, peter).
parent(eve, peter).
parent(adam, paul).
parent(mary, paul).

siblings(U,V):-
 parent(P,U), parent(P,V).
```

Analogamente si possono esprimere le regole per **fratelli, sorelle,** etc.

U è fratello/sorella di V se i due *hanno un genitore P in comune*

TORNANDO ALL'ESEMPIO COI NUMERI..

- Nel paradigma dichiarativo si devono esprimere le relazioni fra le entità, che vanno quindi rappresentate
  - per i numeri naturali *non è opportuno* adottare l'usuale rappresentazione 1,2,3,4.. perché quei simboli *sono legati a un certo significato solo nella nostra testa:* non è una rappresentazione esplicita!
  - in effetti, in matematica: *i numeri naturali* sono definiti come
    - 1 è naturale
    - se N è naturale, anche il *successore* di N lo è
- Seguendo la matematica, in Prolog possiamo ad esempio:
  - indicare con **`s(N)`** il successore di N
  - e con **`eq(X,Y)`** la relazione (equazione) fra x e y, ovvero x = y-2
  - NB: poiché abbiamo in concetto di *successore* ma non quello di predecessore, scriveremo l'equazione nella forma x+2 = y

UN ESEMPIO COL PARADIGMA DICHIARATIVO

- L'equazione x+2 = y si può allora esprimere direttamente come relazione:

```
eq(X,Y) :- s(s(X))=Y.
```

Si legge: **`eq(X,Y)`** è vero **SE** è vero il lato destro, ossia se **`s(s(X))=Y`**

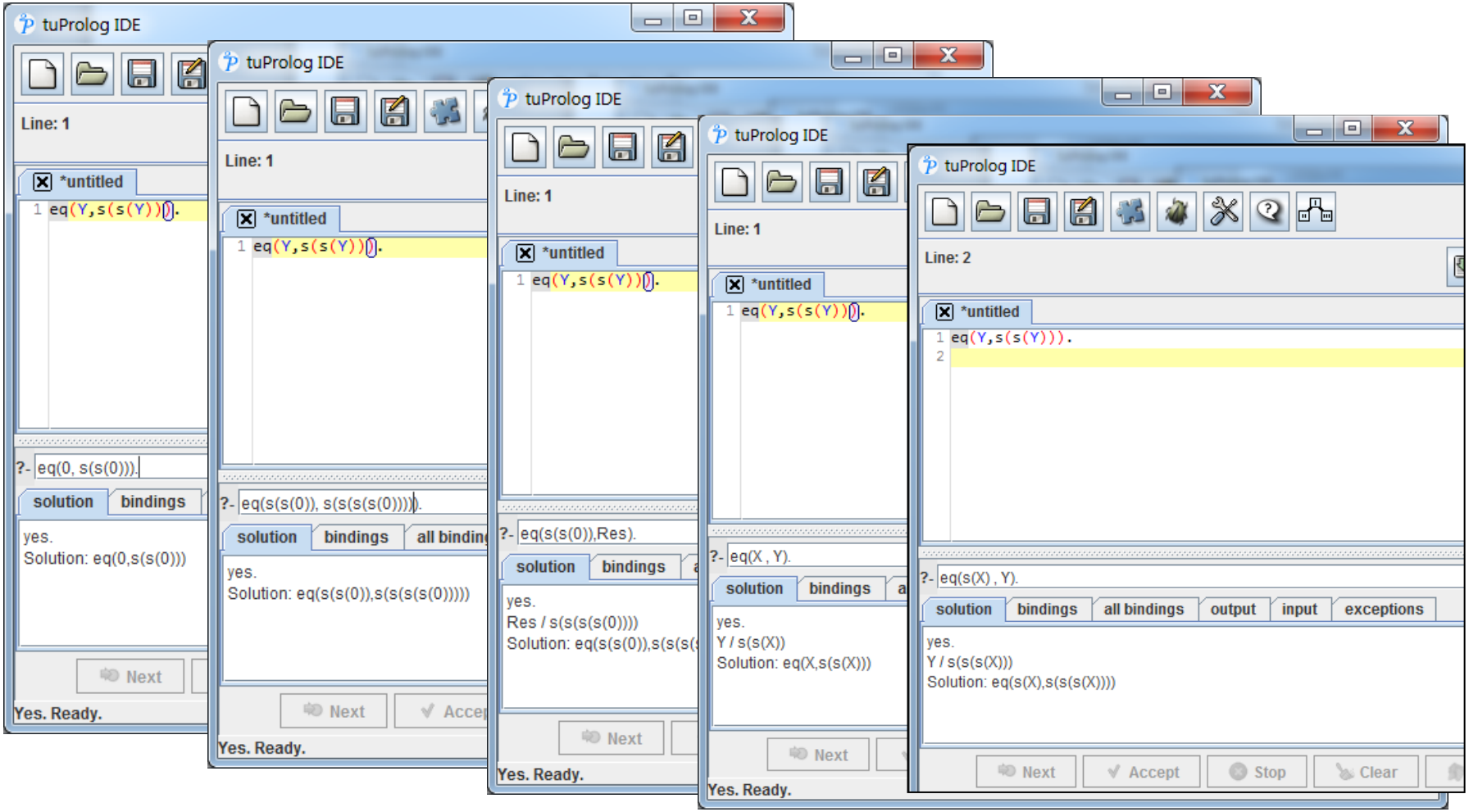
oppure più compattamente:

```
eq(X, s(s(X))).
```

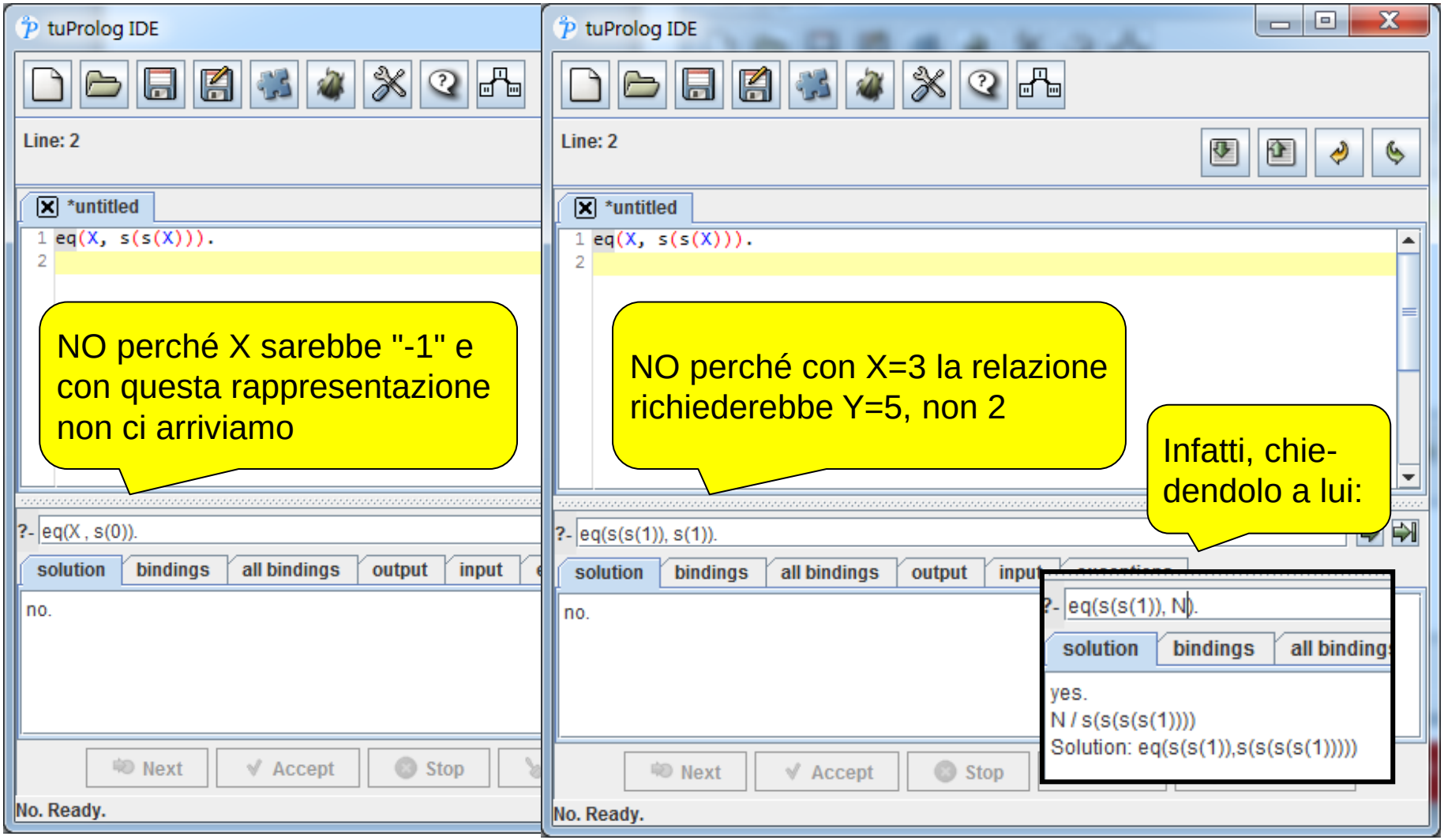
Si legge: **per ogni `X`, è sempre vero** che **`X`** sia in relazione con **`s(s(X))`**

- Ora si può *interrogare il sistema Prolog*
  - fornendo coppie di elementi, ottenendo risposte SI/NO
  - ma anche fornendo uno dei due elementi, ottenendo L'ALTRO
  - o perfino senza fornire nessuno dei due elementi, ottenendo in risposta *la relazione stessa che li lega:* **`Y=s(s(X))`**
  - NB: il caso base può essere 1 o un altro naturale (useremo 0) e si può indicare con qualunque stringa: **`1`**, **`pippo`**, **`uno`**, **`aaa`**,…

UN ESEMPIO COL PARADIGMA DICHIARATIVO



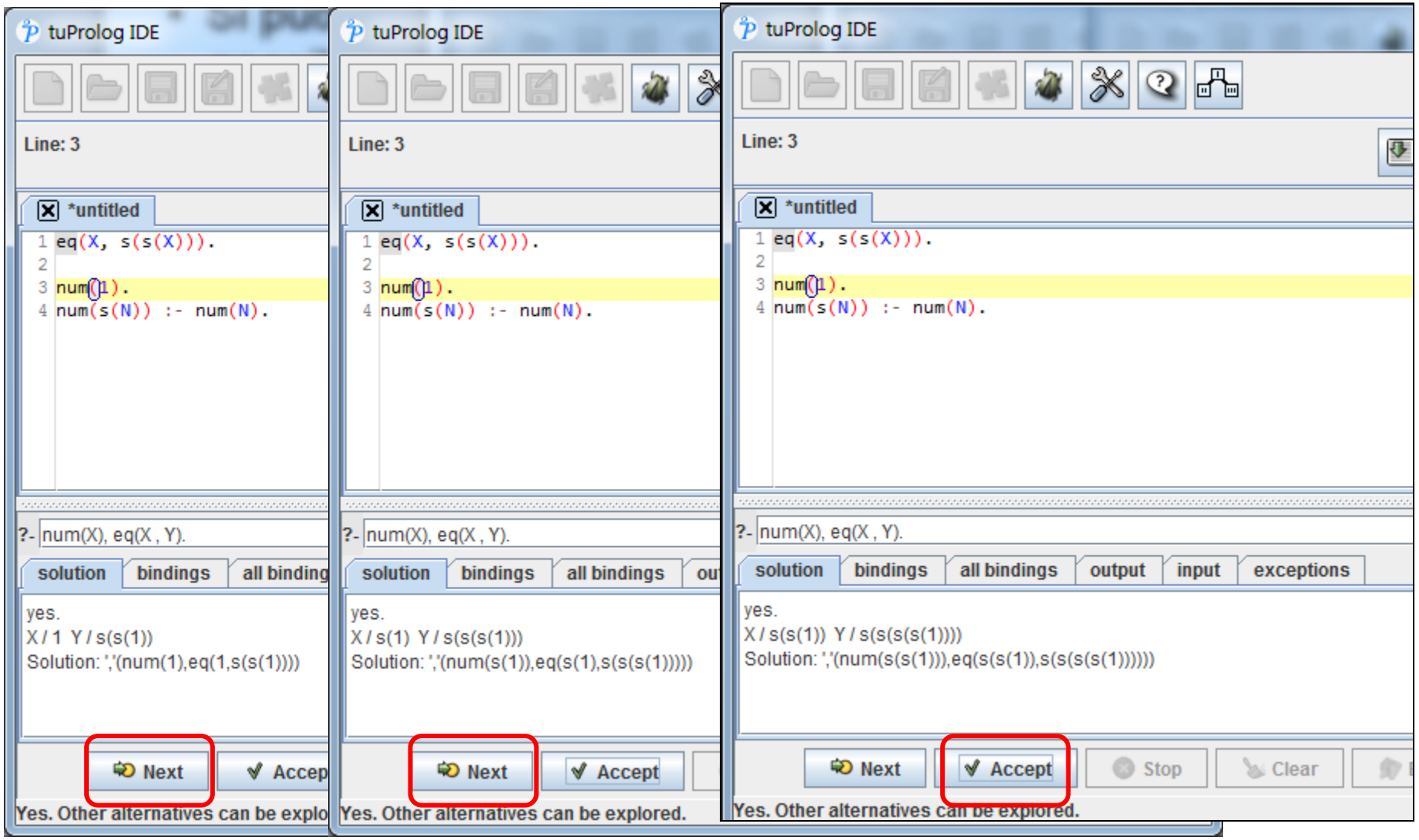
UN ESEMPIO
COL PARADIGMA DICHIARATIVO



UN ESEMPIO COL PARADIGMA DICHIARATIVO

- Si può sfruttare la relazione anche per *far generare via via le possibili soluzioni, una ad una*
- Per farlo bisogna *alimentare* l'equazione con numeri naturali via via diversi
  - ci serve un generatore di numeri naturali
  - per ottenerlo, basta seguire pari pari la definizione matematica:
    - 1 è naturale `num(1).`
    - se N è naturale, anche il *successore* di N lo è `num(s(N)) :- num(N).`
  - invocando `num(X)` si otterranno via via tutte le soluzioni, ossia `1, s(1), s(s(1)), ...`

UN ESEMPIO
COL PARADIGMA DICHIARATIVO

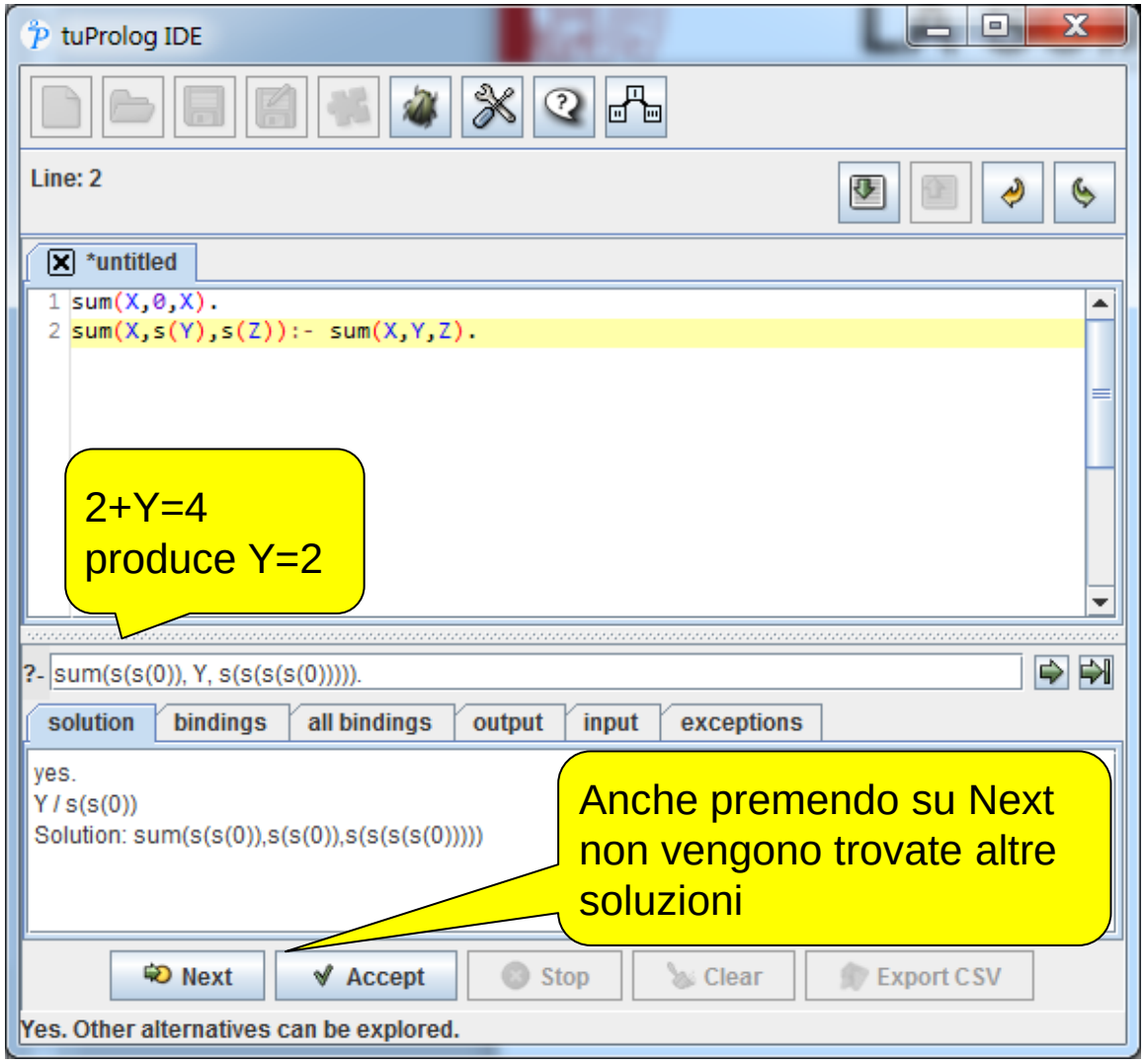


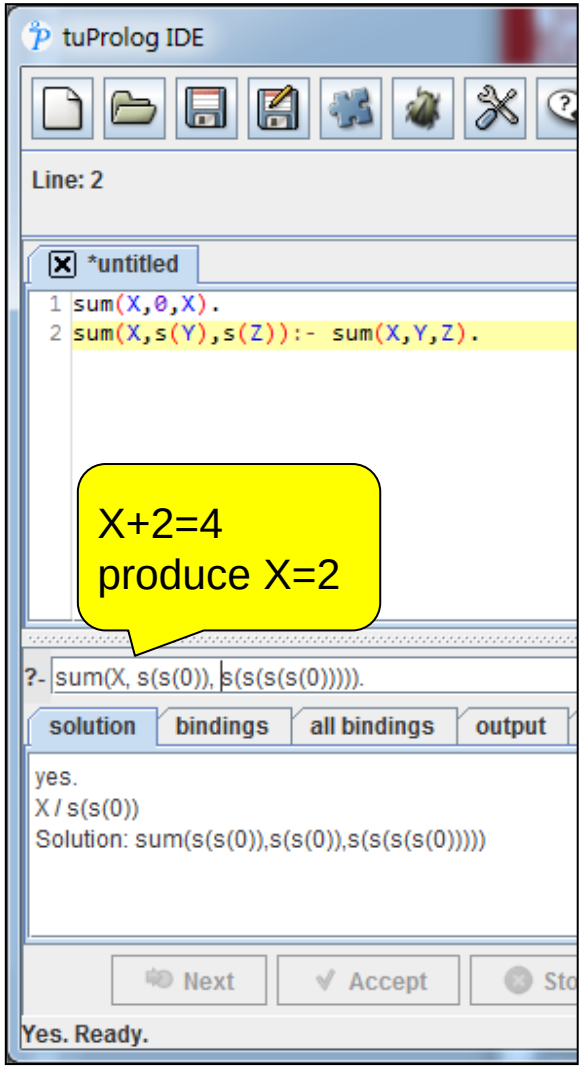
UN ALTRO ESEMPIO
LA SOMMA DI DUE NUMERI

- La somma di due numeri, Z=X+Y, si può trattare nello stesso modo
- Imperativamente, occorrerebbe distinguere i tre casi
  - X,Y in input, Z da calcolare: Z=X+Y
  - X,Z in input, Y da calcolare: Y=Z-X
  - Y,Z in input, X da calcolare: X=Z-Y
- Dichiarativamente, basta esprimere la relazione invariante
  - 0 è l'elemento neutro della somma X+0=X
  - se X+Y=Z, allora la somma di uno di essi col successore dell'altro dà il successore del risultato: X+(Y+1)=(Z+1)

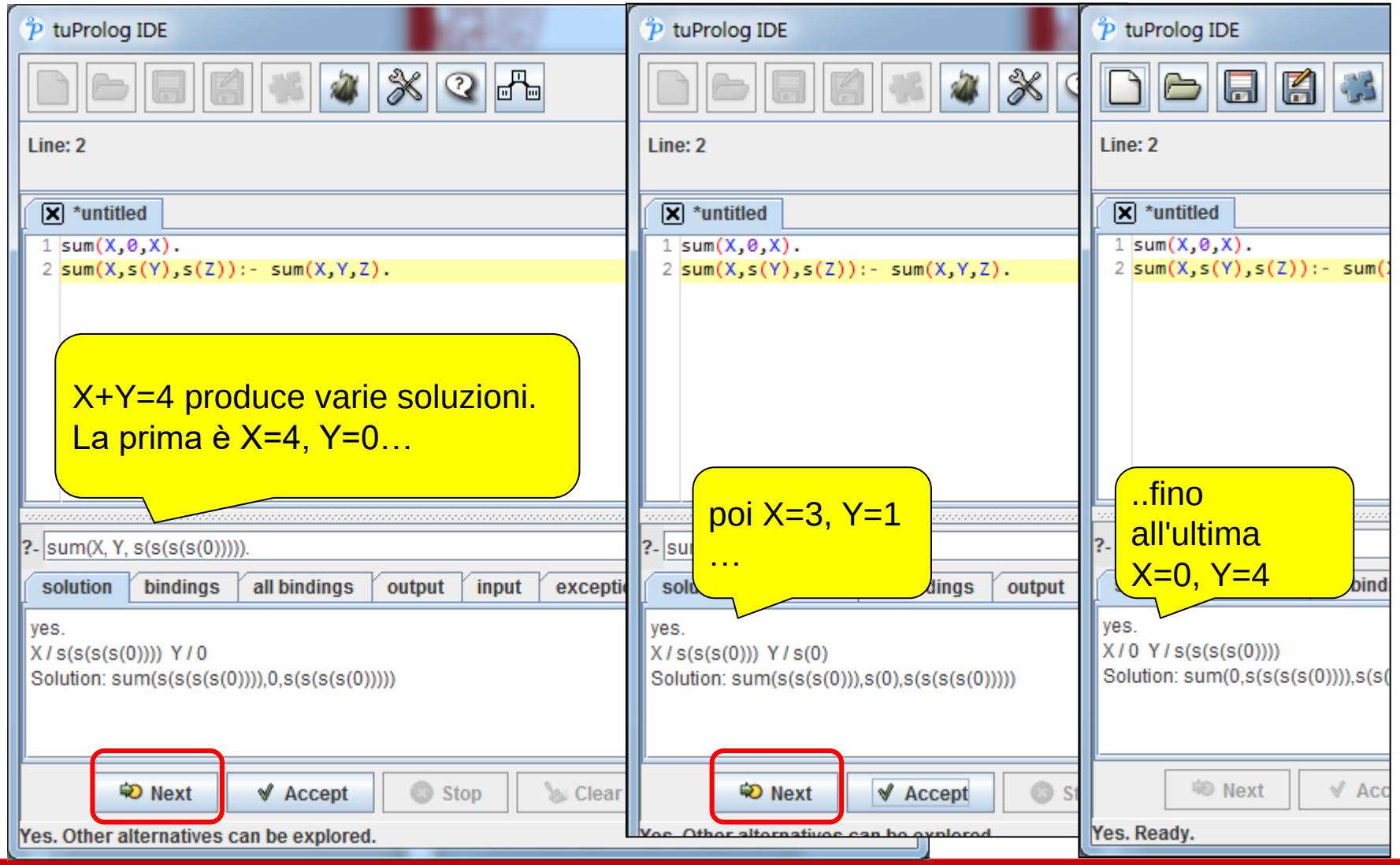
```
sum(X,0,X).
sum(X,s(Y),s(Z)):- sum(X,Y,Z).
```

UN ALTRO ESEMPIO
LA SOMMA DI DUE NUMERI





UN ALTRO ESEMPIO
LA SOMMA DI DUE NUMERI



TRATTAMENTO PRATICO DEI NUMERI IN PROLOG

- In pratica, esprimere i numeri con la notazione successore **`s(N)`** è molto scomodo
- Il linguaggio Prolog accetta quindi numeri reali (non ha la nozione di numero intero) nell'usuale notazione decimale
  - il predicato **`number(X)`** è vero se X rappresenta un reale, ossia ne rispetta la sintassi
  - per calcolare il valore di un'espressione numerica, tuttavia, *non si può usare il classico* = perché in Prolog esso indica unificazione *sintattica:* non ha *semantica*
    - **`Value = 13-4`** dà come risultato la struttura **`13-4`** (ossia, **`'-'(13,4)`**) , non 9

TRATTAMENTO PRATICO DEI NUMERI IN PROLOG

- Per *calcolare* il valore di un'espressione numerica, occorre mettere in pista la semantica dei numeri reali,
- In Prolog essa è embedded nel *predicato speciale* **`is`**
- A differenza di **`Value = 13-4,`** che non effettua alcun «calcolo» perché l'operatore **=** è intenzionalmente privo di semantica specifica di un qualunque dominio,
- **`Value is 13-4`** dà come risultato **`9`**
  perché *prima* di assegnare un valore alla variabile **`Value`** viene fatta una *valutazione semantica* del lato destro, nel dominio dei numeri reali
  - predicato di utilità, extra logico, *non invertibile*

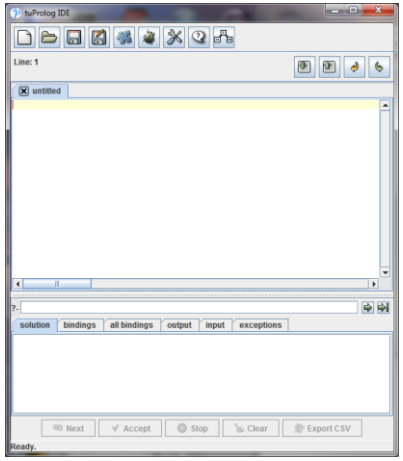
STRUMENTI

- Esistono molti sistemi Prolog, alcuni gratuiti
- SWI-Prolog
- GNU Prolog
- SICStus Prolog (commerciale)
- tuProlog (2p-kt) **`tuprolog.unibo.it`**
  - leggero
  - open source: Apache 2.0
  - multi-piattaforma: Kotlin, JVM, JS, Android
  - ben documentato
  - multi-paradigma: Java/Prolog, Kotlin/Prolog, etc.

tuProlog IN UNA SLIDE

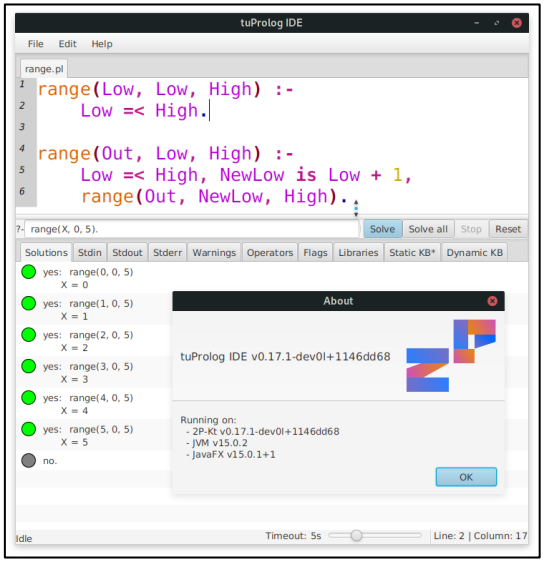
Old 2p classic vs. The new 2p-kt

- Java 8+ - kotlin
- Android 5.x - JVM, JS, Android



Distribuzione old 2p classic

- `2p.jar` contiene tutto
- `2p-lib.jar` contiene solo il motore
- GitHub, Maven central, Docker
- tuprolog.unibo.it



Distribuzione 2p-kt

- https://github.com/tuProlog/2p-kt