



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA

# Prolog – Aritmetica e Ricorsione

**Federico Chesani**

DISI

Department of Informatics – Science and Engineering

# Disclaimer & Further Reading

- These slides are largely based on previous work by Prof. Paola Mello
- Russell Norvig, *AIMA*, vol. 1 ed. italiana: Cap. 9
- Robert Kowalski, "Predicate Logic as a Programming Language", Memo 70, Dept. of Artificial Intelligence, Edinburgh University, 1973. Also in *Proceedings IFIP Congress*, Stockholm, North Holland Publishing Co., 1974, pp. 569–574.
- Lloyd, J. W. (1987). *Foundations of Logic Programming*. (2nd edition). Springer-Verlag.
- Ivan Bratko: *Prolog Programming for Artificial Intelligence*, 4th Edition. Addison-Wesley 2012, ISBN 978-0-3214-1746-6, pp. I-XXI, 1-673
- L. Console, E. Lamma, P. Mello, M. Milano: "Programmazione Logica e Prolog", Seconda Edizione UTET, 1997.
- The Power of Prolog. Introduction to modern Prolog  
<https://www.metalevel.at/prolog>



# Prolog

- Linguaggio **simbolico** (manipolazione di simboli)
- **Non deterministico** (più clausole che definiscono un predicato)
- A priori non ci sono argomenti di ingresso e argomenti di uscita (**invertibilità del codice**), le variabili del top-goal sono le variabili di uscita per cui si cerca un legame (risposta calcolata)



## Prolog – esempio

`arco (a,b) .`

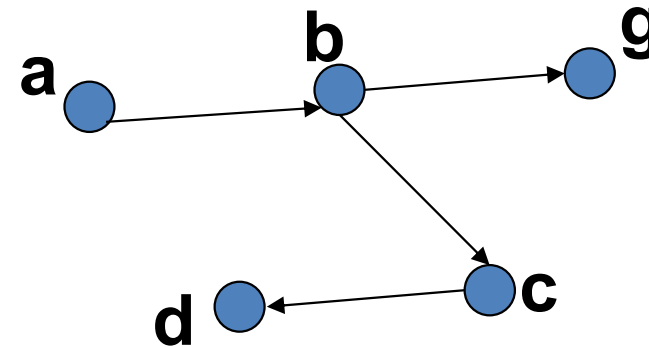
`arco (b,c) .`

`arco (c,d) .`

`arco (b,g) .`

`connesso (X,Y) :- arco (X,Y) .`

`connesso (X,Y) :- arco (X,Z) , connesso (Z,Y) .`



- Tanti modi diversi di "interrogare" il programma:

`?-connesso (a,d) .`

`?-connesso (X,d) .`

`?-connesso (a,Y) .`

`?-connesso (X,Y) .`



# Verso un vero Linguaggio di Programmazione

- Al Prolog puro devono, tuttavia, essere aggiunte alcune caratteristiche per poter ottenere un linguaggio di programmazione utilizzabile nella pratica.
- In particolare:
  - Meccanismi per la definizione e valutazione di espressioni e funzioni.
  - Strutture dati e operazioni per la loro manipolazione.
  - Meccanismi di input/output
  - Meccanismi di controllo della ricorsione e del backtracking.
  - Negazione
- Tali caratteristiche sono state aggiunte al Prolog puro attraverso la definizione di alcuni predicati speciali (**predicati built-in**) predefiniti nel linguaggio e trattati in modo speciale dall'interprete.



# Aritmetica e Ricorsione

- Non esiste, in logica, alcun meccanismo per la **valutazione** di funzioni, operazione fondamentale in un linguaggio di programmazione
- I numeri interi possono essere rappresentati come termini Prolog.
  - Il numero intero  $N$  è rappresentato dal termine:

$s(s(s(\dots s(0)\dots)))$   
*N volte*

```
% prodotto(X, Y, Z) "Z è il prodotto di X e Y"
prodotto(X, 0, 0).
prodotto(X, s(Y), Z) :-    prodotto(X, Y, W),
                           somma(X, W, Z).
```

- **Non utilizzabile in pratica.** Soluzione: **predicati predefiniti per la valutazione di espressioni.**



## Esempio

$2 * 3 - 6$

$- (* (2, 3), 6)$

- Tutto è un termine, in questo linguaggio (anche le clausole stesse del programma)
- Gli operatori infissi sono in realtà simboli di funzione (sebbene particolari e definiti come operatori)



# Predicati Predefiniti per la Valutazione di Espressioni

- L'insieme degli atomi Prolog contiene tanto i numeri interi quanto i numeri floating point. I principali operatori aritmetici sono simboli funzionali (operatori) predefiniti del linguaggio. In questo modo ogni espressione può essere rappresentata come un termine Prolog.
- Per gli operatori aritmetici binari il Prolog consente tanto una notazione prefissa (funzionale), quanto la più tradizionale notazione infissa

**TABELLA OPERATORI ARITMETICI**

|                  |  |
|------------------|--|
| Operatori Unari  | <code>-</code> , <code>exp</code> , <code>log</code> , <code>ln</code> , <code>sin</code> , <code>cos</code> , <code>tg</code> |
| Operatori Binari | <code>+</code> , <code>-</code> , <code>*</code> , <code>\</code> , <code>div</code> , <code>mod</code>                        |

- `+(2,3)` e `2+3` sono due rappresentazioni equivalenti. Inoltre, `2+3*5` viene interpretata correttamente come `2+(3*5)`





# Predicati Predefiniti per la Valutazione di Espressioni

- Data un'espressione, è necessario un meccanismo per la valutazione
- Soluzione: speciale predicato predefinito **is**.

**T is Expr ( is (T, Expr) )**

- **T** può essere un atomo numerico o una variabile
  - **Expr** deve essere un'espressione.
- L'espressione **Expr** viene valutata e il risultato della valutazione viene unificato con **T**

Nota: le variabili in **Expr** DEVONO ESSERE ISTANZIATE al momento della valutazione



# Esempi

```
:- X is 2+3.  
yes      X=5
```

```
:- X1 is 2+3, X2 is exp(X1), X is X1*X2.  
yes      X1=5      X2=148.413      X=742.065
```

```
:- 0 is 3-3.  
yes
```

```
: - X is Y-1.      Y non è istanziata al momento della valutazione!!!!  
No
```

(NOTA: Alcuni sistemi Prolog danno come errore  
Instantion Fault)

```
:- X is 2+3, X is 4+5.  
no
```



## Esempi

```
:- X is 2+3, X is 4+1.  
yes      X=5
```

- In questo caso il secondo goal della congiunzione risulta essere:

```
:-      5 is 4+1.
```

che ha successo. X infatti è stata istanziata dalla valutazione del primo is al valore 5.

```
:-      X is 2+3, X is X+1.  
no
```



NOTA: non corrisponde a un assegnamento dei linguaggi imperativi. **Le variabili sono write-once**



# Esempi

Nel caso dell'operatore `is` l'ordine dei goal è rilevante.

(a)     :-   X is 2+3, Y is X+1.

(b)     :-   Y is X+1, X is 2+3.

- Mentre il goal (a) ha successo e produce la coppia di istanziazioni **X=5**, **Y=6**,
- ... il goal (b) fallisce.
- Il predicato predefinito "is" è un tipico esempio di un predicato **predefinito non reversibile**; come conseguenza le procedure che fanno uso di tale predicato non sono (in generale) reversibili.



# Termini ed Espressioni

- Un termine che rappresenta un'espressione viene valutato solo se è il secondo argomento del predicato **is**

`p(a, 2+3*5) .`

`q(X,Y) :- p(a,Y), X is Y.`

`:- q(X,Y) .`

`yes`   `x=17`   `Y=2+3*5`   `(Y=+(2, *(3,5)))`



Valutazione di Y


NOTA: **Y** non viene valutato, ma unifica con una struttura che ha **+** come operatore principale, e come argomenti **2** e una struttura che ha **\*** come operatore principale e argomenti **3** e **5**



# Operatori Relazionali

- Il Prolog fornisce **operatori relazionali per confrontare i valori di espressioni**.
- Tali operatori sono utilizzabili come goal all'interno di una clausola Prolog ed hanno notazione infissa

## OPERATORI RELAZIONALI

$>$ ,  $<$ ,  $>=$ ,  $=<$ ,  $==$ ,  $=\backslash=$   *disuguaglianza*



*uguaglianza*



# Confronto tra Espressioni

- Passi effettuati nella valutazione di:

**Expr1 REL Expr2**

dove **REL** è un operatore relazionale e **Expr1** e **Expr2** sono espressioni

- vengono valutate **Expr1** ed **Expr2**
- NOTA: **le espressioni devono essere completamente istanziate**
- I risultati della valutazione delle due espressioni vengono confrontati tramite l'operatore **REL**



# Unificazione e uguaglianza tra termini

- **T1 = T2**, verifica se T1 e T2 sono **unificabili**. Viene generata la sostituzione (mgu) che unifica i due termini (e vengono pertanto legate le variabili nei due termini).

?-  $f(X, g(a)) = f(h(c), g(Y))$ .

yes  $X=h(c)$   $Y=a$

?-  $2+3 = 5$ .

no

- **T1 == T2**, verifica se T1 e T2 sono **uguali** (identici). In particolare, se i due termini contengono due variabili in posizioni equivalenti, perché i termini siano uguali, le due variabili devono essere la stessa variabile. Si noti che in questo caso non viene generata alcuna sostituzione.

?-  $f(a, b) == f(a, b)$ . Yes

?-  $f(a, X) == f(a, b)$ . No

?-  $f(a, X) == f(a, X)$ . Yes

?-  $f(a, X) == f(a, Y)$ . no





# Uguaglianza tra espressioni (SICStus Prolog e SWIsh Prolog)

- **T1 == T2**, verifica se i termini T1 e T2 sono **uguali** (identici). Non li valuta anche se rappresentano espressioni

?- 2\*2 == 4.

no

- **E1 ::= E2**, verifica se E1 e E2 sono espressioni che hanno lo stesso valore

?- 2\*2 ::= 4.

yes



# Disuguaglianza (SICStus Prolog e SWIsh Prolog)

- **T1 \== T2**, verifica se T1 e T2 non sono uguali (identici); ha successo se i due termini (non valutati) non sono identici  
?- a\==b.      **yes**
- In SWIsh (e anche SWI, e SICtus) Prolog per le espressioni, l'operatore diverso è indicato come: **=\=**
- **E1 =\= E2** ha successo se le due espressioni (che vengono valutate) non hanno lo stesso valore
- Quindi
  - $2 * 2 \backslash = 4$       è vero
  - $2 * 2 = \backslash 4$       è falso



# Calcolo di Funzioni

- Una funzione può essere realizzata attraverso relazioni Prolog.
- Data una funzione  $f$  ad  $n$  argomenti, essa può essere realizzata mediante un predicato ad  $n+1$  argomenti nel modo seguente

–  $f : x_1, x_2, \dots, x_n \rightarrow y$  diventa  
 $f(x_1, x_2, \dots, x_n, Y) \text{ :- } \langle \text{calcolo di } Y \rangle$

Esempio: calcolare la funzione fattoriale così definita:

`fatt: n → n ! (n intero positivo)`

`fatt(0) = 1`

`fatt(n) = n * fatt(n-1) (per n>0)`

`fatt(0,1).`

`fatt(N,Y) :- N>0, N1 is N-1, fatt(N1,Y1), Y is N*Y1.`



# Calcolo di Funzioni

Esempio: calcolare il massimo comun divisore tra due interi positivi

`% definizione:`

`mcd:  $x, y \rightarrow \text{MCD}(x, y)$  ( $x, y$  interi positivi)`

`$\text{MCD}(x, 0) = x$`

`$\text{MCD}(x, y) = \text{MCD}(y, x \bmod y)$  (per  $y > 0$ )`

`% mcd(X, Y, Z)`

`% "Z è il massimo comun divisore di X e Y"`

`mcd(X, 0, X) .`

`mcd(X, Y, Z) :- Y > 0, X1 is X mod Y,  
mcd(Y, X1, Z) .`



# Esempi

- Calcolare la funzione

```
% abs(x) = |x|  
% abs(X,Y)  
% "Y è il valore assoluto di X"  
abs(X,X) :- X >= 0.  
abs(X,Y) :- X < 0, Y is -X.
```

- Si consideri la definizione delle seguenti relazioni:

```
% pari(X) = true se X è un numero pari  
% false se X è un numero dispari  
% dispari(X) = true se X è un numero dispari  
% false se X è un numero pari
```

```
pari(0).
```

```
pari(X) :- X > 0, X1 is X-1, dispari(X1).
```

```
dispari(X) :- X > 0, X1 is X-1, pari(X1).
```



# Ricorsione e Iterazione

- Il Prolog non fornisce alcun costrutto sintattico per l'iterazione (quali, ad esempio, i costrutti while e repeat) e l'unico meccanismo per ottenere iterazione è la definizione ricorsiva.
- Una funzione  $f$  è definita per **ricorsione tail** se  $f$  è la funzione "più esterna" nella definizione ricorsiva o, in altri termini, se sul risultato della chiamata ricorsiva di  $f$  non vengono effettuate ulteriori operazioni
- La definizione di funzioni (predicati) per ricorsione tail può essere considerata come una definizione per iterazione
  - Potrebbe essere valutata in spazio costante mediante un processo di valutazione iterativo.



# Ricorsione e Iterazione

- Si dice **ottimizzazione della ricorsione tail** valutare una funzione tail ricorsiva  $f$  mediante un processo iterativo ossia caricando un solo record di attivazione per  $f$  sullo stack di valutazione (esecuzione).
- In Prolog l'ottimizzazione della ricorsione tail è un po' più complicata che non nel caso dei linguaggi imperativi a causa del:
  - non determinismo
  - della presenza di punti di scelta nella definizione delle clausole.



# Ricorsione e Iterazione

$p(X) :- c1(X), g(X).$   
(a)  $p(X) :- c2(X), h1(X,Y), p(Y).$   
(b)  $p(X) :- c3(X), h2(X,Y), p(Y).$

- Due possibilità di valutazione ricorsiva del goal : -  $p(Z)$ .
  - se viene scelta la clausola (a), si deve ricordare che (b) è un punto di scelta ancora aperto. Bisogna mantenere alcune informazioni contenute nel record di attivazione di  $p(Z)$  (i punti di scelta ancora aperti)
  - se viene scelta la clausola (b) (più in generale, l'ultima clausola della procedura), non è più necessario mantenere alcuna informazione contenuta nel record di attivazione di  $p(Z)$  e la rimozione di tale record di attivazione può essere effettuata





## Quindi...

- In Prolog l'ottimizzazione della ricorsione tail è possibile solo se la scelta nella valutazione di un predicato "p" è deterministica o, meglio, se al momento della chiamata ricorsiva (n+1)-esima di "p" non vi sono alternative aperte per la chiamata al passo n-esimo (ossia alternative che potrebbero essere considerate in fase di backtracking)
- **Quasi tutti gli interpreti Prolog effettuano l'ottimizzazione della ricorsione tail ed è pertanto conveniente usare il più possibile ricorsione di tipo tail.**



# Ricorsione Tail

- Il predicato **fatt** è definito con una forma di ricorsione semplice (non tail).
- Casi in cui una relazione ricorsiva può essere trasformata in una relazione tail ricorsiva

```
fatt1(N,Y) :- fatt1(N,1,1,Y) .
```

```
fatt1(N,M,ACC,ACC) :- M > N.
```

```
fatt1(N,M,ACCin,ACCout) :- ACCtemp is ACCin*M,  
                           M1 is M+1,  
                           fatt1(N,M1,ACCtemp,ACCout) .
```

*Accumulatore  
in ingresso*

*Accumulatore  
in uscita*



# Ricorsione Tail

- Il fattoriale viene calcolato utilizzando un argomento di accumulazione, inizializzato a 1, incrementato ad ogni passo e unificato in uscita nel caso base della ricorsione.

$$ACC_0 = 1$$

$$ACC_1 = 1 * ACC_0 = 1 * 1$$

$$ACC_2 = 2 * ACC_1 = 2 * (1 * 1)$$

...

$$ACC_{N-1} = (N-1) * ACC_{N-2} = N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots))$$

$$ACC_N = N * ACC_{N-1} = N * (N-1 * (N-2 * (\dots * (2 * (1 * 1)) \dots)))$$



# Ricorsione Tail

- Altra struttura iterativa per la realizzazione del fattoriale

`fatt2(N,Y)    "Y è il fattoriale di N"`

`fatt2(N,Y)    :-   fatt2(N,1,Y) .`

`fatt2(0,ACC,ACC) .`

`fatt2(M,ACC,Y)    :-   ACC1 is M*ACC,  
                         M1 is M-1,  
                         fatt2(M1,ACC1,Y) .`



# Ricorsione Non-Tail

- Calcolo del numero di Fibonacci: definizione

```
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(N) =
    fibonacci(N-1) + fibonacci(N-2)    per N > 1
```

- Programma Prolog:

```
fib(0,0) .
fib(1,1) .
fib(N,Y)  :-      N>1,
                  N1 is N-1,
                  fib(N1,Y1),
                  N2 is N-2,
                  fib(N2,Y2),
                  Y is Y1+Y2.
```



# E il classico Input/Output?

- **read(X)**
  - Legge un termine da console
- **write(X)**
  - Stampa X (il termine a cui è legata X) a video
- **nl**
  - Stampa un "a capo" (newline)



# Few exercises



## Few instructions...

- When the prolog interpreter is executed, it assumes a “working directory”: it will look for files only in the current working directory.
  - To discover the current working dir: “**pwd.**”
  - To change working dir: “**working\_directory(Old, New) .**”
- Programs must be loaded, through the pre-processing of source files
  - Given a file “**test.pl**”, it can be loaded through “**consult(test) .**”
- When a program source is modified, it needs to be reloaded into the database clause
  - Command “**make.**” reloads all the changed files
  - It is always a good practice to close the interpreter, and run it again, from time to time...
- Debugging?
  - Command “**trace.**” ... but before, read the official documentation...

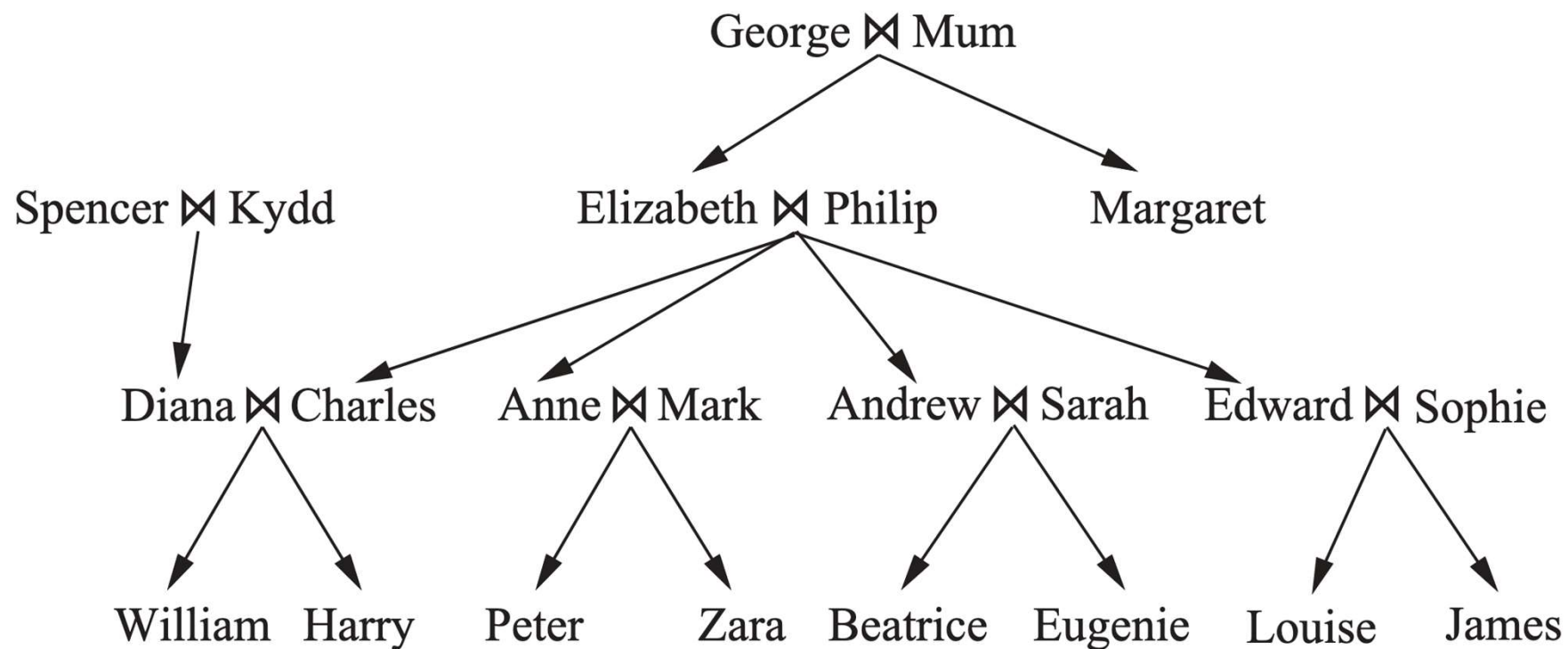




## Exercise – the kinship domain

(Sect, 8.3.2 and Ex. 8.kins from ALMA)

Represent the family relations as function terms and predicates.



A typical family tree. The symbol  $\bowtie$  connects spouses and arrows point to children.



## Exercise – the kinship domain

“Concepts” we would like to represent:

- Male
- Female
- Parent
- Father
- Mother
- Sibling
- Brother
- Sister
- Child
- Daughter
- Son
- Spouse
- Wife
- Husband



## Exercise – the kinship domain

“Concepts” we would like to represent:

- Granparent
- Grandchild
- Cousin
- Aunt
- Uncle
- Greatgrandparent
- Ancestor
- FirstCousin
- BrotherInLaw
- SisterInLaw



## Excercise

Define a Prolog program that receives in input a number N, and print all the numbers between 1 and N.

Solution:

```
specialPrint(1) :-  
    write(1), nl.  
specialPrint(N) :-  
    N>1,  
    write(N), nl,  
    TheNext is N-1,  
    specialPrint(TheNext) .
```



## Excercise

Define a Prolog program that receives in input a number N, and print all the numbers between 1 and N, from the smallest to the greatest.

Solution:

```
specialPrint(1) :-  
    write(1), nl.  
specialPrint(N) :-  
    N>1,  
    TheNext is N-1,  
    specialPrint(TheNext),  
    write(N), nl.
```



## Excercise

Write a predicate about a number N, that is true if N is prime

Solution:

```
isPrime(2).
```

```
isPrime(N) :-
```

```
    N>2,
```

```
    TheDiv is N-1,
```

```
    cannotBeDividedBy(N, TheDiv).
```

```
cannotBeDividedBy(N, 2) :-
```

```
    1 is N mod 2.
```

```
cannotBeDividedBy(N, TheDiv) :-
```

```
    TheDiv>2,
```

```
    Rest is N mod TheDiv,
```

```
    Rest > 0,
```

```
    TheNextDiv is TheDiv-1,
```

```
    cannotBeDividedBy(N, TheNextDiv).
```



## Excercise

Write a predicate that, given a number N, prints out all the prime numbers between 2 and N.

Solution:

```
printOnlyPrimes(2) :-  
    printOnlyIfItIsPrime(2).  
printOnlyPrimes(N) :-  
    N>2,  
    printOnlyIfItIsPrime(N),  
    TheNext is N-1,  
    printOnlyPrimes(TheNext).
```

```
printOnlyIfItIsPrime(N) :-  
    isPrime(N),  
    write('The number '), write(N), write(' is prime!'), nl.  
printOnlyIfItIsPrime(N) :-  
    \+isPrime(N),  
    write('The number '), write(N), write(' is NOT prime!'), nl.
```

