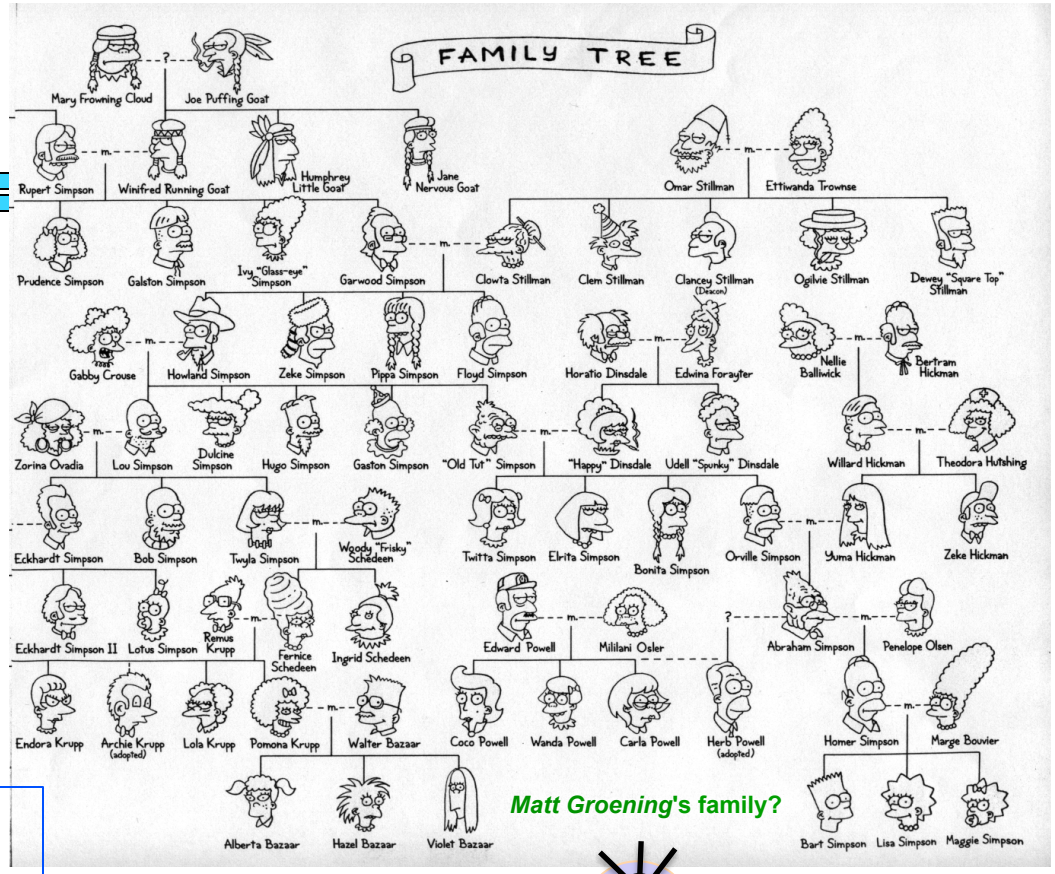
# Today in CS 42

**yes**

**Hw #10: Prolog**




***The Simpsons* character names** [edit]

Groening famously named the main Simpson characters after members of his own family: his parents, Homer and Margaret (Marge or Marjorie in full), and his younger sisters, Lisa and Margaret (Maggie).

Coming up…

Monday 11/14: HW10 due (Prolog intro)

Thursday 11/18: **Midterm distributed**

Monday 11/21: NO HW DUE

Tuesday 11/22: Midterm due

Thursday 11/24: Thanksgiving (no class)

Tuesday 11/29: HW11 due (Prolog games)

# Review: Tail Recursion and Hmmm

```
(define (tower N)
  (tower-tr N 2))

(define (tower-tr N A)
  (if (= N 0)
      A
      (power A (tower-tr (- N 1) A))))
```

Is tower-tr tail recursive?

A. Yes

B. No

C. Not sure

# Review: Tail Recursion and Hmmm

```
(define (power-tr b n A)
  (if (= n 0)
      A
      (power-tr b (- n 1) (* A b))))
```

```
42 jeqz r2 46
43 addn r2 -1
44 mul r3 r3 r1
45 jump 42
46 mov r13 r3
47 jumpi r14
```

```
42 jeqz r2 51
43 addn r2 -1
44 mul r3 r3 r1
45 storei r14 r15
46 addn r15 1
47 call r14 42
48 addn r15 -1
49 loadi r14 r15
50 jumpi r14
51 mov r13 r3
52 jumpi r14
```

Which is a "faithful" implementation of the Racket code above in HMMM?

A. left side only

B. right side only

C. both

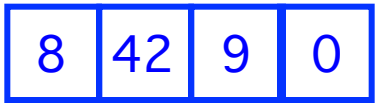
D. neither

# Big O comparison

Please do not try these at home!

Problem

Find the maximum element in a list




paranoid algorithm

**First, verify the < operator with 2,000 known comparisons (1 < 2 )**
**Second, look at each list element, compare it with the current max**
**-- Replace the current max if necessary.**
**As you do this, make each comparison 1,000 times just to "be sure"**

code reuse algorithm

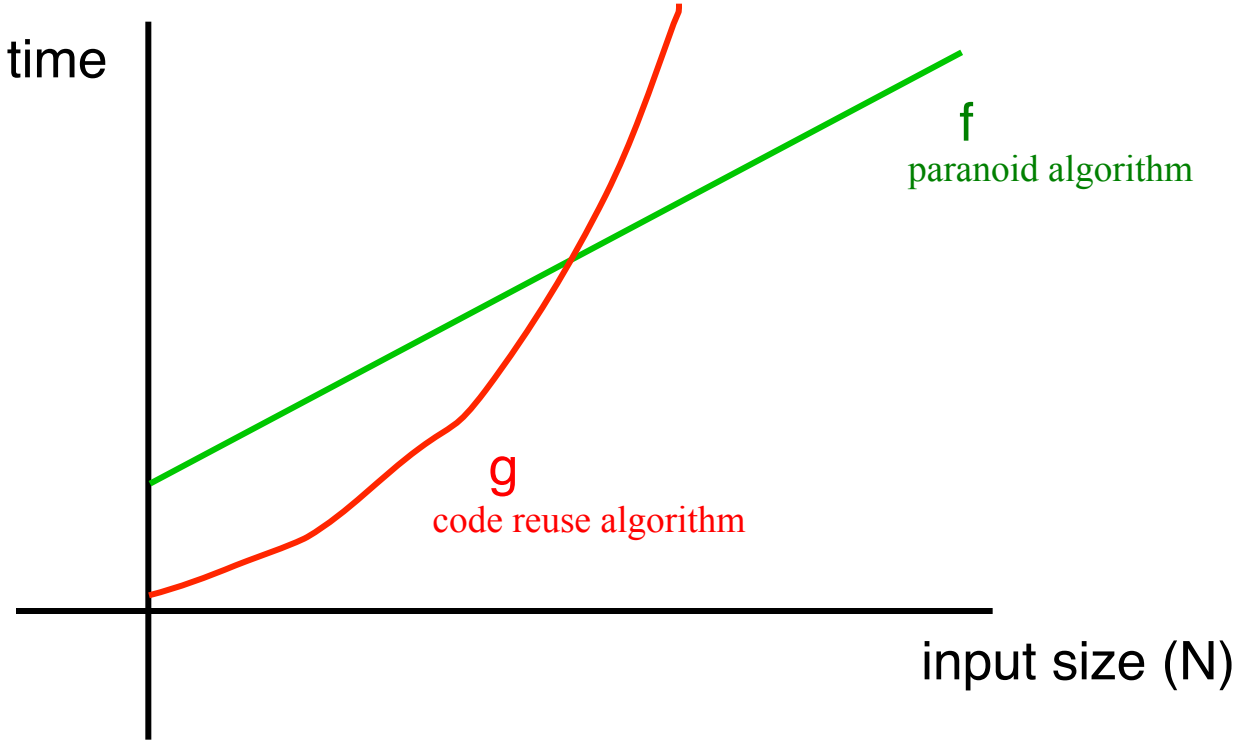
**You have a minimum-element finder already.**
**All elements start unmarked.**
**Find the smallest unmarked element and mark it.**
**Repeat, keeping track of the last element marked each time.**
**When they're all marked, return the last element marked.**

# The Big Picture on “Big Oh”

**O** (“Big Oh”) compares running times for algorithms

f is O(g)

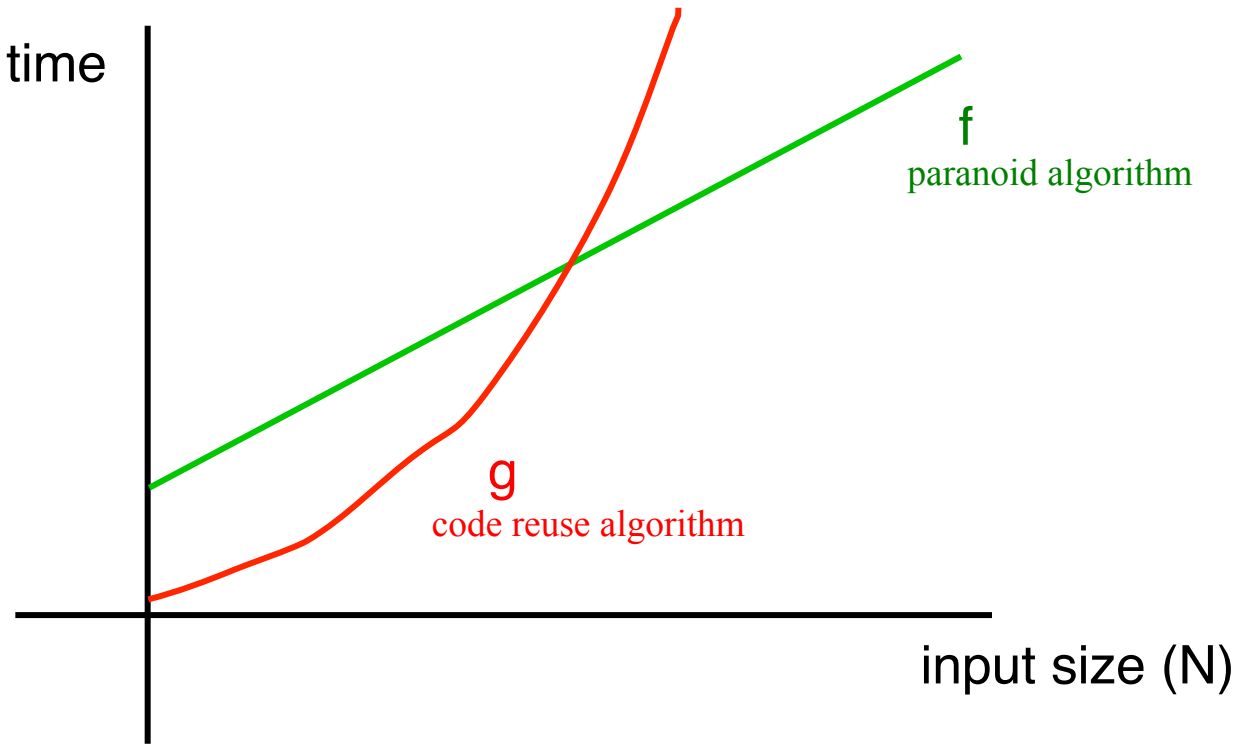
g is not O(f)



# The Formal Definition of Big-O

f(n) is O(g(n)) if there exists constants c and $n_0$ s.t.

$f(n) \leq c*g(n)$ for all $n \geq n_0$



# Big **O** in practice

big-O = hand-waving math!

Ignore everything but the dominant term

even there, ignore the constant

$$420n^2 + 3n^3 - 201$$

$$17 + 300\sqrt{n} + 0.5\log(n)$$

$$\frac{117}{n} + \frac{2^n}{3^n} + 1700$$

Big-O is simply an upper bound, but normally we want the **BEST** upper bound…

# Review: Big-O

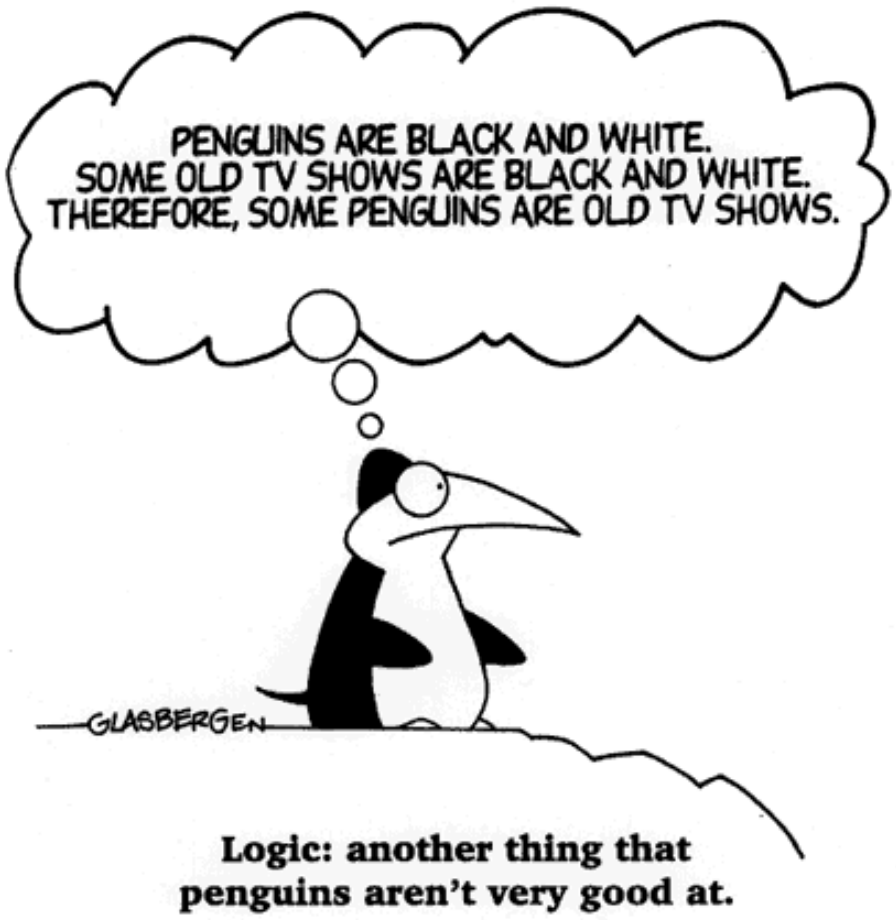
What is Big-O running time of the following mystery function? (and what does it do?)

```
(define (mystery N)
  (reverse (mystery-help N)))

(define (mystery-help N)
  (cond
    ((= N 0) '())
    ((= 1 (modulo N 2)) (cons 1 (mystery-help (/ (- N 1) 2))))
    (else (cons 0 (mystery-help (/ N 2))))))
```

A. O(log N)  B. O(N)  C. O(N log N)  D. All of these  E. None of these

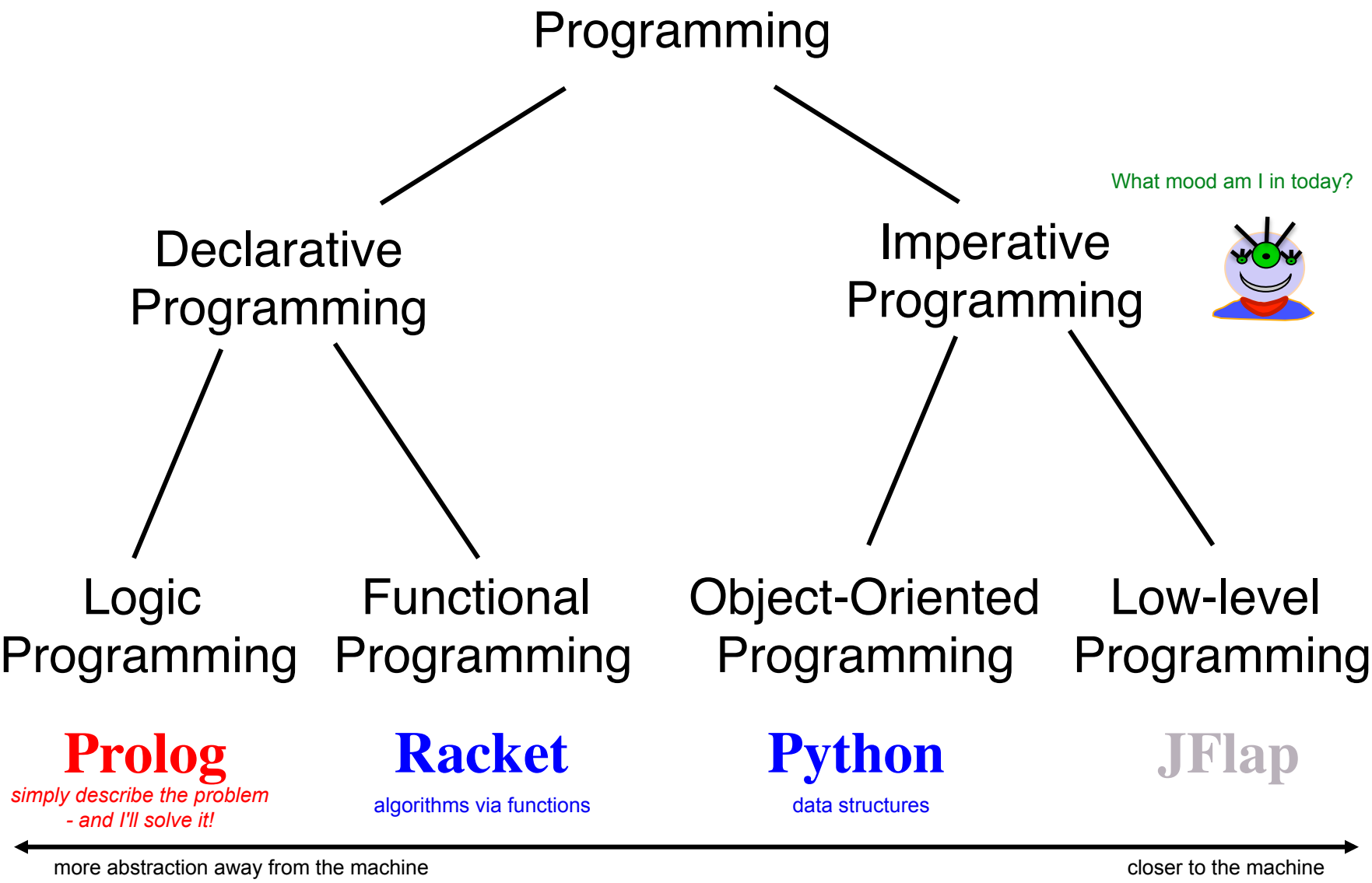
# Midterm #2

- Take home, distributed Thursday Nov 17
- Same ground rules as first midterm.
- Everything up through Prolog (this week)
  even if we only saw it in class.
  including topics from before midterm 1
- Short reviews over the next two lectures
  More review problems posted online soon




# And now, for something completely different…

# Taxonomy of Programming Models



# Prolog: The Dream

We feed Prolog a bunch of facts relevant to our problem

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
pairs(apple, banana).
pairs(banana, ginger).
pairs(banana, cloves).
pairs(banana, strawberry).
pairs(banana, coriander).
pairs(strawberry, honey).
pairs(strawberry, ginger).
pairs(strawberry, tea).
pairs(tea, walnut).
pairs(tea, tomato).
pairs(tea, milk).

pairs(X,X).
pairs(X, coconut).
```

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us.

So buy now! **http://www.swi-prolog.org/**

Demo…

# Prolog: Just the Facts!

```
%% Here is a comment in Prolog
/* this is also a comment */

chocolate_is_good.   %% A fact, but not in a very useful form

good(spam).  %% a debatable fact, but now prolog thinks so
good(42).    %% no one can deny this…

weird_list([3.14,42,spam,'+',"hi"]).  %% Well, I think so.


better(tofu, spam).          %% "tofu" is better than spam
better(chocolate, tofu).     %% "chocolate" is better than "tofu"
better(money, chocolate).    %% "money" is better than "chocolate"


%% Does prolog know that money is better than spam?  Does prolog even
%% know what spam is???  NO!
```

# Prolog can be alien…


- To run: `/opt/local/bin/swipl` **(macs)**
  `swipl` **(ssh to knuth.cs.hmc.edu server)**
- To exit: `halt.`
- To (re)load a file: `[file].` or `reconsult('file.pl').`

`[user].` lets you type facts

- The Prolog environment is for queries only!
  Use your favorite editor to change files…

Windows's `swipl` has an editor built-in.

```
parent(homer,bart).
parent(homer,lisa).
parent(marge,bart).
parent(marge,lisa).

child(A,B) :- parent(B,A).
```

FACTS

RULES

some facts and rules from the file `simpsons.pl`

```
?:  parent(homer,lisa).
yes

?:  parent(X,bart).
X = homer  ;
X = marge  ;
no
```

TEST

GENERATE

queries within the Prolog environment

# Prolog syntax is the stranger

you mean L'Étranger…

**The idea:** Provide data/relationships that describe the world. Then ask questions.

**Relationships:**

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

child(X, Y) :- parent(Y, X).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

colon-hyphen means "IF"

variables always start with a capital letter

commas mean "AND"

predicates and constants start with a lowercase letter

these relationships are ***predicates*** not functions. They *have* a value, but ***they do not return a value!***

multiple rules express "OR"
; also means "OR"

recursion welcome!

# File **simpsons.pl**

```
/*
 * the parent predicate
 */

parent(olf, skug).
parent(helga, skug).
parent(skug, esmerelda).
parent(skugerina, esmerelda).
parent(esmerelda, klotho).
parent(gemini, klotho).
parent(esmerelda, atropos).
parent(gemini, atropos).
parent(esmerelda, lachesis).
parent(gemini, lachesis).
parent(olf, homericus).
parent(helga, homericus).
parent(ug, matilda).
parent(uggette, matilda).
parent(homericus, homer).
parent(matilda, homer).
parent(homericus, gomer).
parent(matilda, gomer).
parent(homer, bart).
parent(marge, bart).
parent(homer, lisa).
parent(marge, lisa).
parent(homer, maggie).
parent(marge, maggie).
parent(john, marge).
parent(jackie, marge).
parent(john, selma).
parent(jackie, selma).
parent(john, patty).
parent(jackie, patty).
parent(john, glum).
parent(jackie, glum).
parent(glum, millhouse).
parent(cher, millhouse).
parent(glum, wentworth).
parent(cher, wentworth).
```

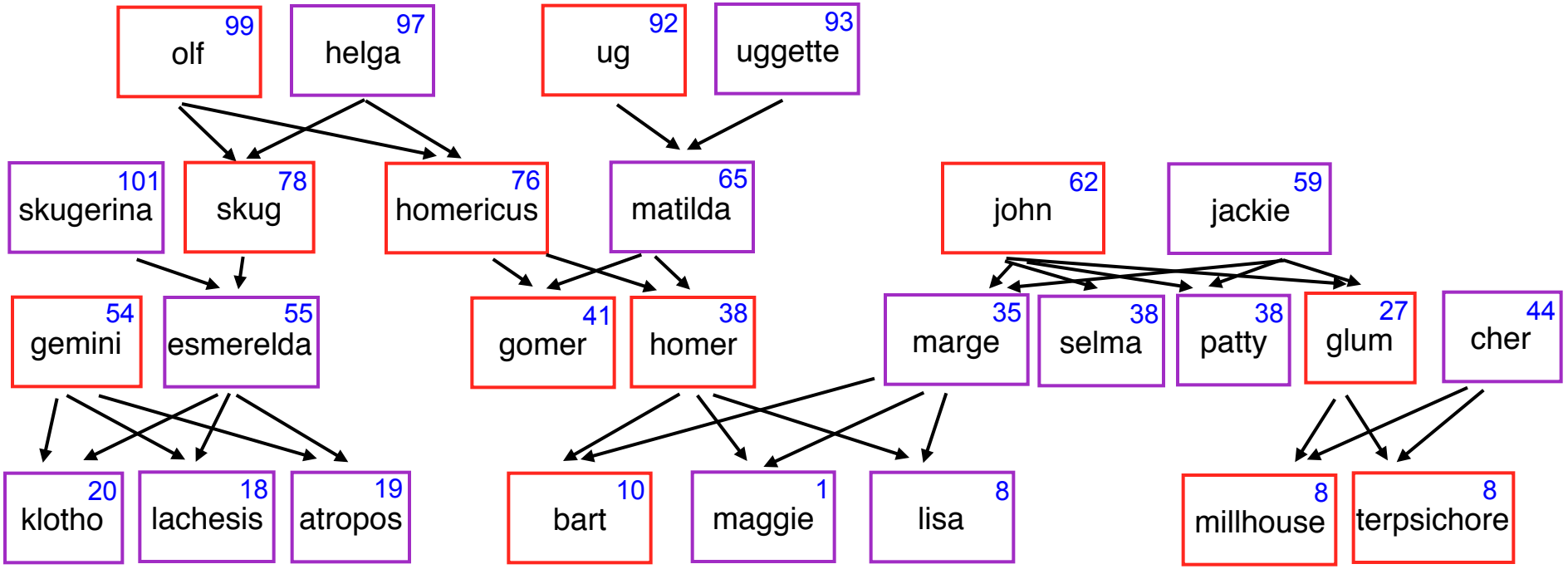
```
/*
 * the age predicate
 */

age(helga, 97).
age(olf, 99).
age(uggette, 93).
age(ug, 92).
age(matilda, 65).
age(homericus, 76).
age(skugerina, 101).
age(skug, 78).
age(esmerelda, 55).
age(gemini, 54).
age(klotho, 20).
age(atropos, 19).
age(lachesis, 18).
age(marge, 35).
age(homer, 38).
age(lisa, 8).
age(maggie, 1).
age(bart, 10).
age(gomer, 41).
age(john, 62).
age(jackie, 59).
age(patty, 38).
age(selma, 38).
age(glum, 27).
age(cher, 44).
age(millhouse, 8).
age(wentworth, 8).
```

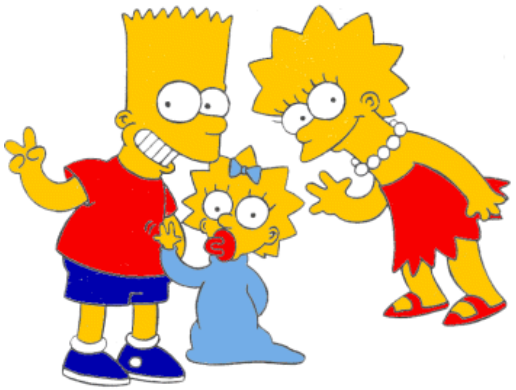
```
/*
 * the female predicate
 */

female(helga).
female(esmerelda).
female(skugerina).
female(uggette).
female(matilda).
female(marge).
female(jackie).
female(selma).
female(patty).
female(cher).
female(lisa).
female(maggie).
female(klotho).
female(atropos).
female(lachesis).
```

```
/*
 * the male predicate
 */

male(olf).
male(skug).
male(homericus).
male(ug).
male(homer).
male(gomer).
male(gemini).
male(john).
male(glum).
male(bart).
male(millhouse).
male(wentworth).
```

```
/*
 * Here are three rules about families
 */

child(X, Y) :- parent(Y, X).

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

anc(X, Y) :- parent(X, Y).
anc(X, Y) :- parent(Z, Y), anc(X, Z).
```

# A more complete picture ...



The idea:

Provide data/relationships that describe the world. Then ask questions.

Questions:

child(lisa,marge).

**Is Lisa a child of Marge?**

parent(marge,X).

**Who has Marge as a parent?**

**Is Ug a parent of Bart?**

**Who are the ancestors of Lisa?**

**Who are uggette's descendants?**

**Who is who's parent?**

**Who are John's grandchildren?**

Name:

# “Quiz”

Write these three family-relationship predicates in prolog:

---

**`sibs(X,Y)`**

true if X and Y are siblings

---

**`aunt(A,N)`**

true if A is N’s aunt

---

**`rel(X,Y)`**

true if X and Y are related (by blood)

Name:

# "Quiz"

Are these definitions correct? A. Yes B. No

---

**sibs(X,Y)** true if X and Y are siblings

```
sibs( X, Y ) :- parent(A,X), parent(A,Y).
```

---

**aunt(A,N)** true if A is N's aunt

```
aunt( X, Y ) :-  parent( Q, N ), sibs( Q, A ), female( A ).
```

---

**rel(X,Y)**
true if X and Y are related (by blood)

```
rel( X, Y ) :- anc( Z, X ), anc( Z, Y ).
```

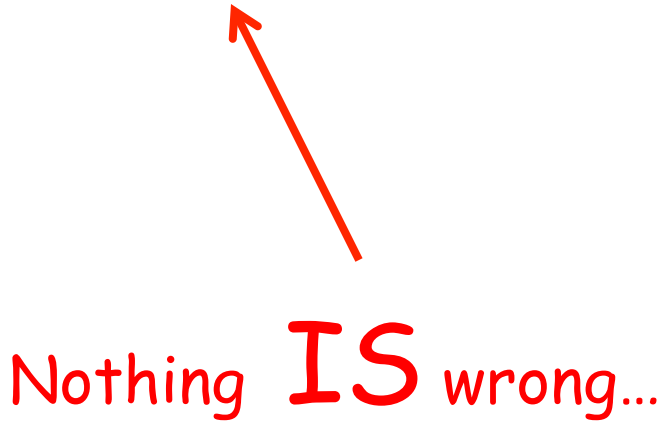
# What's wrong here?

```
sibs(X, Y)  :-  parent (A,X),
                parent (A,Y),
                X \== Y.
```

Nothing is wrong…

# What's wrong here?

```
sibs(X, Y) :-  parent (A,X),
               parent (A,Y),
               X \== Y.
```



# setof

```
?- setof(A,aunt(A,bart),S).
?- setof(N,aunt(marge,N),S).
```

setof(R, p(R), S)

is true when S is the set (list) of Rs that satisfy predicate p.

trace. / notrace. / debug. / nodebug.

spy(female).

nospyall.

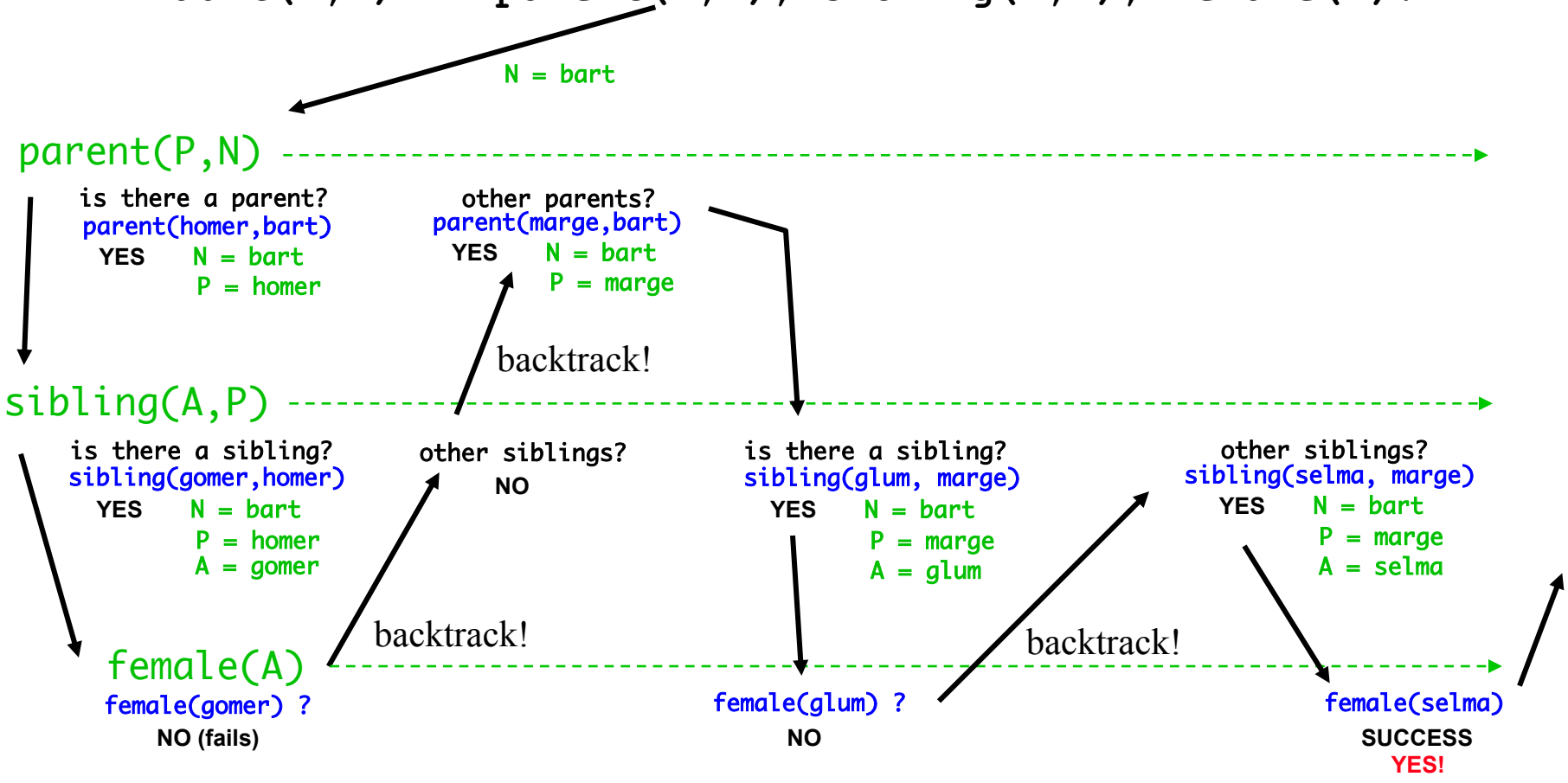
# Prolog `is` DFS

Maybe this should be Prolog == DFS?

Who are Bart's aunts?

aunt(A,bart)

```
aunt(A,N) :- parent(P,N), sibling(A,P), female(A).
```



# Prolog: The Dream

We feed Prolog a bunch of facts relevant to our problem

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avacado).
pairs(walnut, banana).
pairs(apple, banana).
pairs(banana, ginger).
pairs(banana, cloves).
pairs(banana, strawberry).
pairs(banana, coriander).
```

```
pairs(strawberry, honey).
pairs(strawberry, ginger).
pairs(strawberry, tea).
pairs(tea, walnut).
pairs(tea, tomato).
pairs(tea, milk).

pairs(X,X).
pairs(X, coconut).
```

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us.

So buy now! **http://www.swi-prolog.org/**

Demo…

# Prolog: The Reality

We feed Prolog a bunch of facts relevant to our problem

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%% etc.
```

```
pairs(X,X).
pairs(X, coconut).

pairs(X,Y) :- pairs(Y,X).
```

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us?

In practice, you have to think hard about DFS...

Infinite loop Demo…

# Prolog: The Reality (2)

We feed Prolog a bunch of facts relevant to our problem

```
pairs(X,Y) :- pairs(Y,X).

pairs(X,X).
pairs(X, coconut).
```

```
pairs(apple, walnut).
pairs(apple, honey).
pairs(walnut, avocado).
pairs(walnut, banana).
%% etc.
```

We describe how to *recognize* a solution

```
yummy_triple(X,Y,Z) :- pairs(X,Y),  X \= Y,
                       pairs(Y,Z),  Y \= Z,
                       pairs(X,Z),  X \= Z.
```

Prolog then finds solution(s) for us?

In practice, you often have to think hard about DFS...

Infinite Loop Demo…