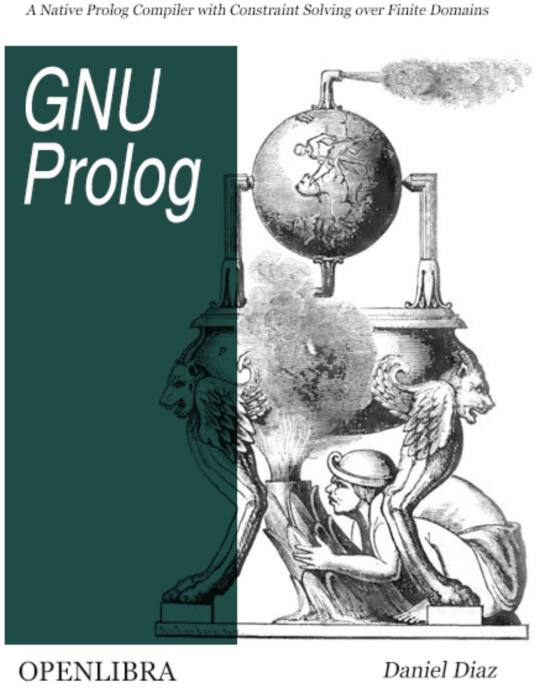
CS 4100: Introduction to AI

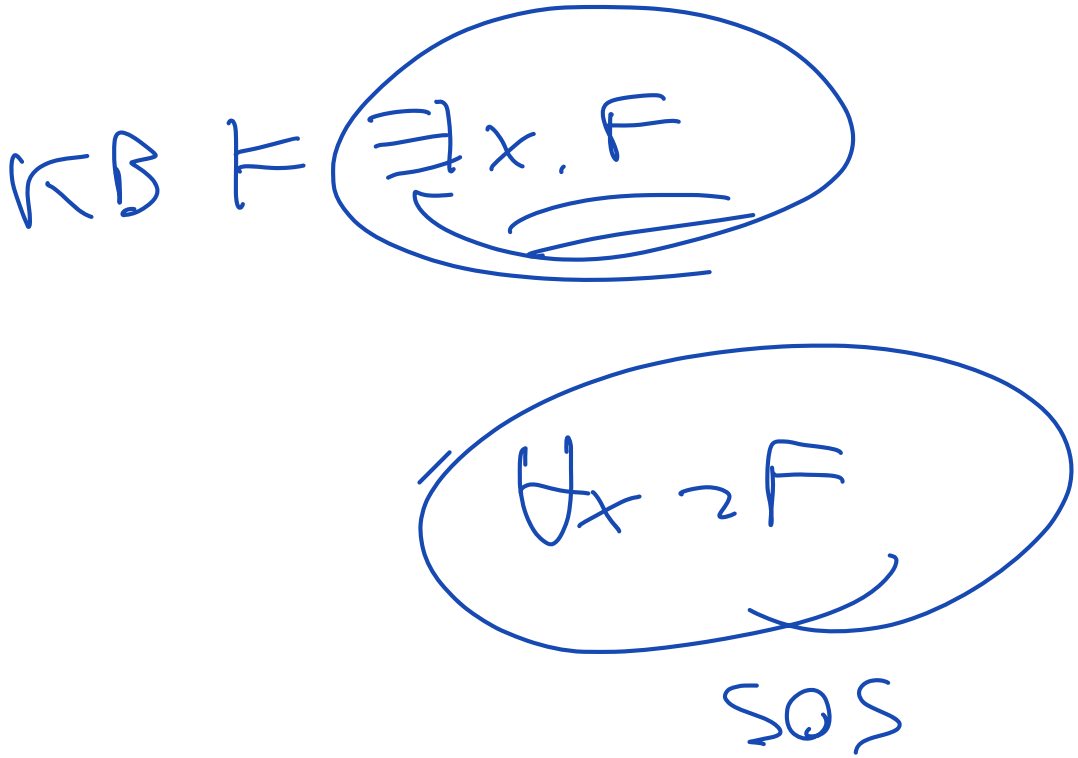
Wayne Snyder
Northeastern University

Lecture 6: Conclusion to Theorem Proving in FOL; Prolog



Resolution Theorem Proving in FOL

(Demonstration of FOL Resolution system)



Prolog: PROgramming in Logic

Prolog is a programming language based on resolution theorem proving in first-order logic, first developed around 1972 by Alain Colmerauer and Robert Kowalski.

Colmerauer

Prolog was widely use in AI during the 1980s, including major research efforts:

- European Espirit
- ICOT Fifth Generation Computer Systems Initiative

Kowalski

Currently it is used in academia and in several current systems, including IBM's Watson, as part of the NLP subsystem. The developers have said that

"We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness."

Prolog: PROgramming in Logic

Prolog is a programming language based on resolution theorem proving in first-order logic, first developed around 1972 by Alain Colmerauer and Robert Kowalski.

Colmerauer

Prolog was widely use in AI during the 1980s, including major research efforts:

- European Espirit
- ICOT Fifth Generation Computer Systyems Initiative

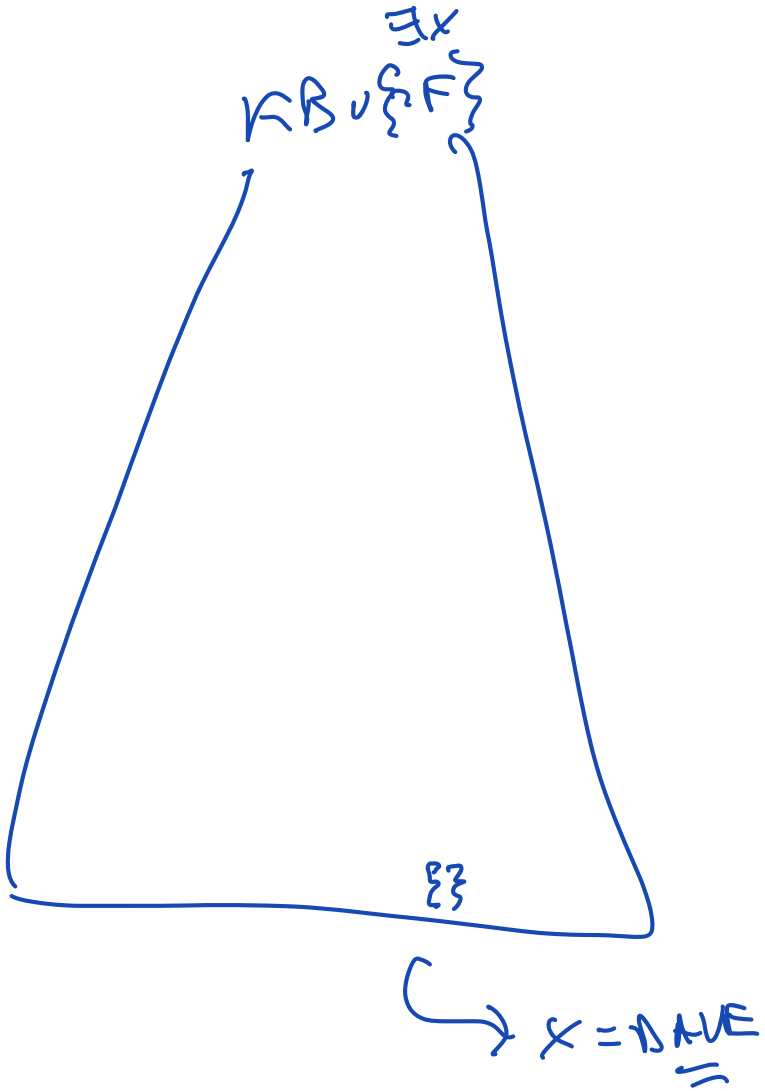
Kowalski

Currently it is used in academia and in several current systems, including IBM's Watson, as part of the NLP subsystem. The developers have said:

"We required a language in which we could conveniently express pattern matching rules over the parse trees and other annotations (such as named entity recognition results), and a technology that could execute these rules very efficiently. We found that Prolog was the ideal choice for the language due to its simplicity and expressiveness."

Prolog: How to program in FOL?

What are the main differences between a search program like our FOL prover and a programming language?



CONS(1, CONS(2, NIL))

[1, 2]

Prolog: How to program in FOL?

Non-determinism (hard to predict execution steps) and inefficiency are mitigated by using a restricted form of formula (Horn clauses) and a restricted form of resolution (SLD resolution).

Recall: Horn clauses have at most a single positive literal:

FACTS

$Female(isabelle)$

$Female(karen)$

$Child(isabelle, anne, oscar)$

$Child(oscar, karen, franz)$

RULES

$\forall x \forall y \forall z.\ Offspring(x, y) \vee \neg Child(x, y, z)$

$\forall x \forall y \forall z.\ Offspring(x, z) \vee \neg Child(x, y, z)$

$\forall x \forall y.\ Parent(y, x) \vee \neg Offspring(x, y)$

$\forall x \forall y.\ Mother(x, y) \vee \neg Female(x) \vee \neg Parent(x, y)$

$\forall x \forall y.\ Grandmother(z, x) \vee \neg Mother(z, y) \vee \neg Offspring(x, y)$

GOAL

$\forall x.\ \neg Grandmother(karen, x)$

SOS

$\vee$ OR

$\{MAN(x), WOMAN(x)\}$

$\{CHILD(x), ADULT(x)\}$

2^n

Prolog: Horn Clauses

It is MUCH clearer to write these as implications (when there are negative literals), and leave out the quantifiers:

$Female(isabelle)$

$Female(karen)$

$Child(isabelle, anne, oscar)$

$Child(oscar, karen, franz)$

$\forall x \forall y \forall z.\ \ Offspring(x, y) \vee \neg Child(x, y, z)$

$\forall x \forall y \forall z.\ \ Offspring(x, z) \vee \neg Child(x, y, z)$

$\forall x \forall y.\ \ Parent(y, x) \vee \neg\ Offspring(x, y)$

$\forall x \forall y.\ \ Mother(x, y) \vee \neg Female(x) \vee \neg Parent(x, y)$

$\forall x \forall y.\ \ Grandmother(z, x) \vee \neg Mother(z, y) \vee \neg\ Offspring(x, y)$

$\forall x.\ \neg Grandmother(karen, x)$

$Female(isabelle)$

$Female(karen)$

$Child(isabelle, anne, oscar)$

$Child(oscar, karen, franz)$

$Offspring(x, y) \Leftarrow Child(x, y, z)$

$Offspring(x, z) \Leftarrow Child(x, y, z)$

$Parent(y, x) \Leftarrow\ Offspring(x, y)$

$Mother(x, y) \Leftarrow Female(x) \wedge Parent(x, y)$

$Grandmother(z, x) \Leftarrow Mother(z, y) \wedge\ Offspring(x, y)$

$\neg Grandmother(karen, x)$

Note that what we have done is to move the positive literal to the left of $\Leftarrow$ and negatives to the right, which makes the logical connection much clearer: "X is the mother of Y if X is female and X is a parent of Y."

Analogously, we can express the query ("set of support") as an implication with no "head":

$$() \Leftarrow Grandmother(karen, x)$$

Prolog: Horn Clauses

Finally, in order to write these as plain text, we use **:-** for $\Leftarrow$ and use a period to show the end of the statement.

$Female(isabelle)$

$Female(karen)$

$Child(isabelle, anne, oscar)$

$Child(oscar, karen, franz)$

$Offspring(x, y) \Leftarrow Child(x, y, z)$

$Offspring(x, z) \Leftarrow Child(x, y, z)$

$Parent(y, x) \Leftarrow Offspring(x, y)$

$Mother(x, y) \Leftarrow Female(x) \wedge Parent(x, y)$

$Grandmother(z, x) \Leftarrow Mother(z, y) \wedge Offspring(x, y)$

$\neg Grandmother(karen, x)$

```
Female(isabelle).
Female(karen).
Child(isabelle, anne, oscar)
Child(oscar, karen, franz).
Offspring(x, y) :- Child(x, y, z).
Offspring(x, z) :- Child(x, y, z).
Parent(y, x) :- Offspring(x, y).
Mother(x, y) :- Female(x), Parent(x, y).
Grandmother(z, x) :- Mother(z, y), Offspring(x, y).
?- Grandmother(karen, x).
```

PROLOG PROGRAM

RULES ARE ORDERED (LISTS)

QUERY

GOAL CLAUSE.

?-

{ ?- PARENT(X,Y), MALE(X) }

?- PARENT(X,Y), MALE(X)

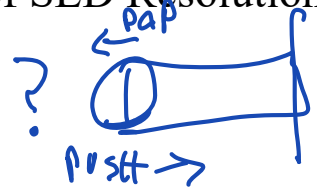
Prolog: SLD Resolution

SLD = "Selection Rule Linear Derivation":

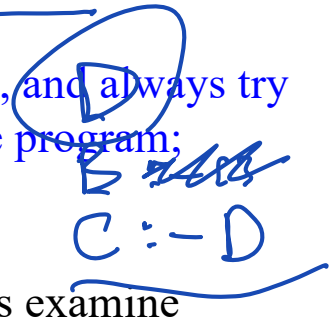
Linear Resolution means that there is only ONE clause (the "Goal Clause") that we resolve against, and

a Selection Rule tells us which literal in the goal clause to resolve on.

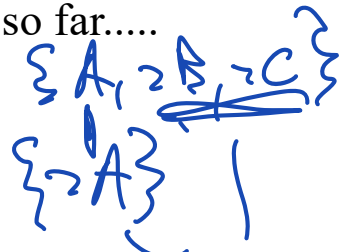
PROLOG is a particular implementation of SLD Resolution with the following features:



- We use a stack instead of a queue (so we are doing depth-first search);
- We start (as usual) by putting the goal clause on the stack;
- We treat the program as a list of clauses, and always try rules/facts in the order they occur in the program;



There are other "bells and whistles" but let's examine what we have so far.....



Female(*isabelle*).

Female(*karen*).

Child(*isabelle*, *anne*, *oscar*)

Child(*oscar*, *karen*, *franz*).

Offspring(*x*, *y*) :- *Child*(*x*, *y*, *z*).

Offspring(*x*, *z*) :- *Child*(*x*, *y*, *z*).

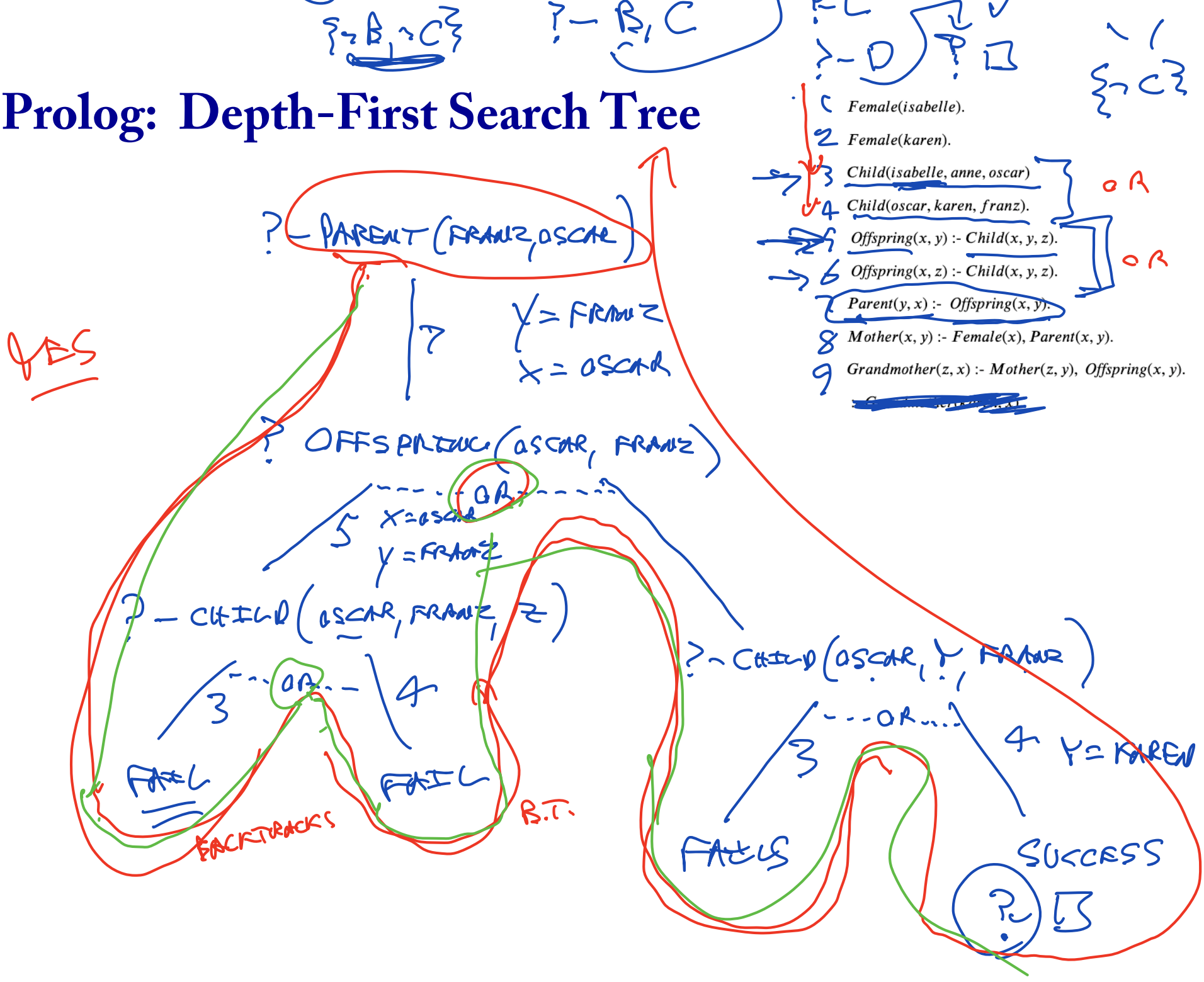
Parent(*y*, *x*) :- *Offspring*(*x*, *y*).

Mother(*x*, *y*) :- *Female*(*x*), *Parent*(*x*, *y*).

Grandmother(*z*, *x*) :- *Mother*(*z*, *y*), *Offspring*(*x*, *y*).

:- *Grandmother*(*karen*, *x*).

Prolog: Depth-First Search Tree



Prolog: Depth-First Search Tree

Female(*isabelle*).

Female(*karen*).

Child(*isabelle*, *anne*, *oscar*)

Child(*oscar*, *karen*, *franz*).

Offspring(*x*, *y*) :- *Child*(*x*, *y*, *z*).

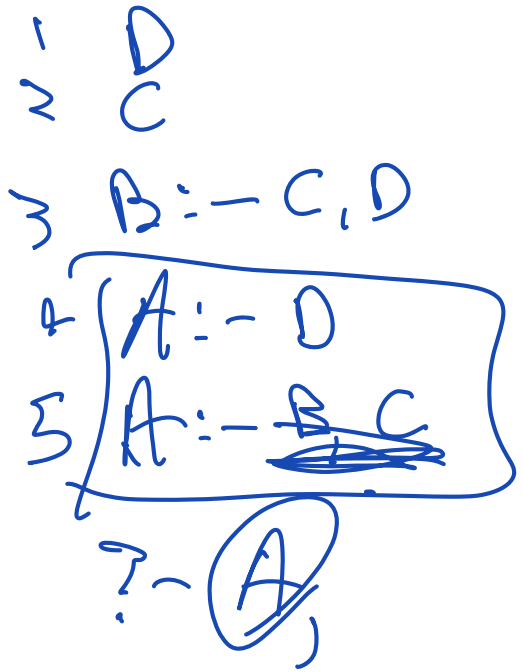
Offspring(*x*, *z*) :- *Child*(*x*, *y*, *z*).

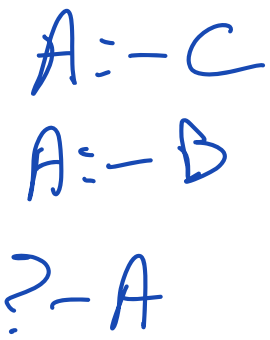
Parent(*y*, *x*) :- *Offspring*(*x*, *y*).

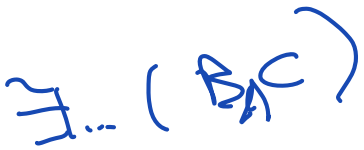
Mother(*x*, *y*) :- *Female*(*x*), *Parent*(*x*, *y*).

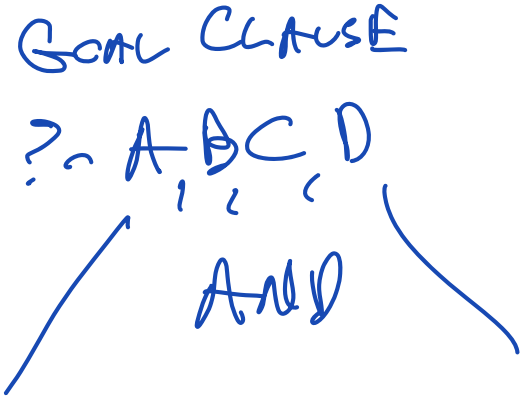
Grandmother(*z*, *x*) :- *Mother*(*z*, *y*), *Offspring*(*x*, *y*).

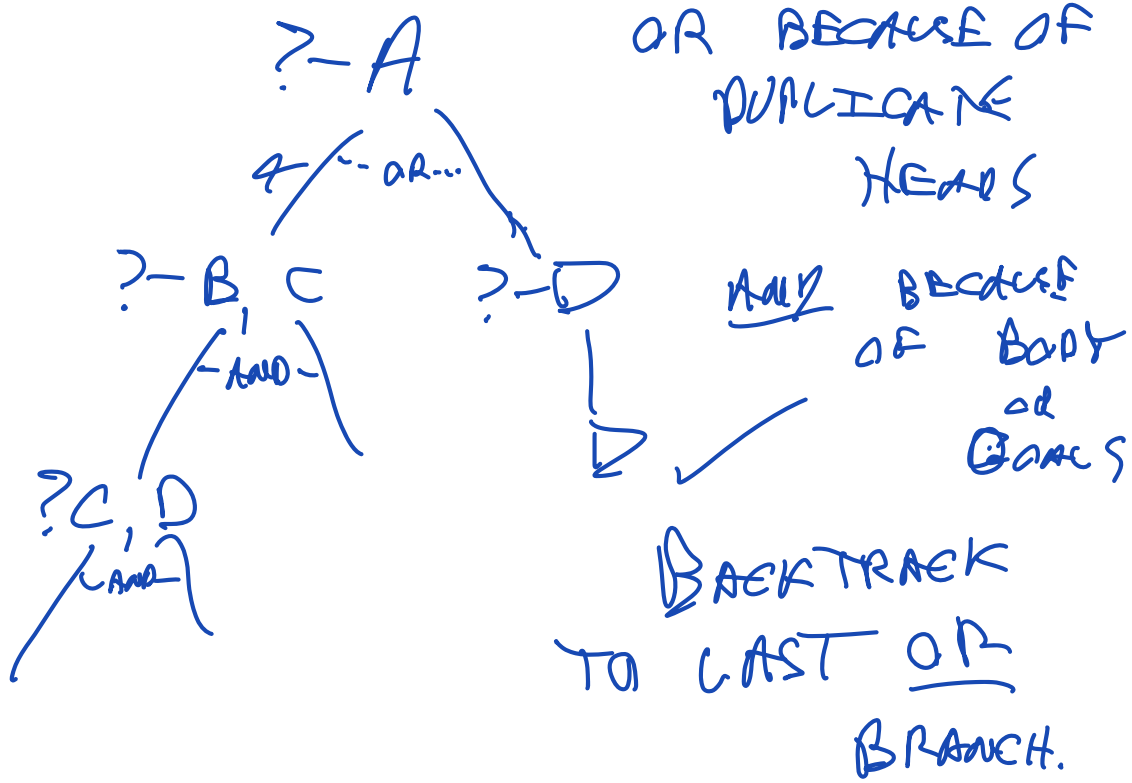
:- *Grandmother*(*karen*, *x*).



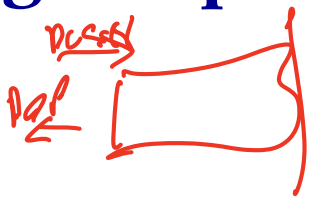








Prolog: Depth-First Search Tree



1. *Female(isabelle).*
1. *Female(isabelle).*
2. *Female(karen).*
3. *Child(isabelle, anne, oscar).*
4. *Child(oscar, karen, franz).*
5. *Offspring(x, y) :- Child(x, y, z).*
6. *Offspring(x, z) :- Child(x, y, z).*
7. *Parent(y, x) :- Offspring(x, y).*
8. *Mother(x, y) :- Female(x), Parent(x, y).*
9. *Grandmother(z, x) :- Mother(z, y), Offspring(x, y).*

:-

:- *Grandmother(karen, x).*

?- MOTHER(KAREN, X)

?- FEMALE(KAREN), PARENT(KAREN, X)

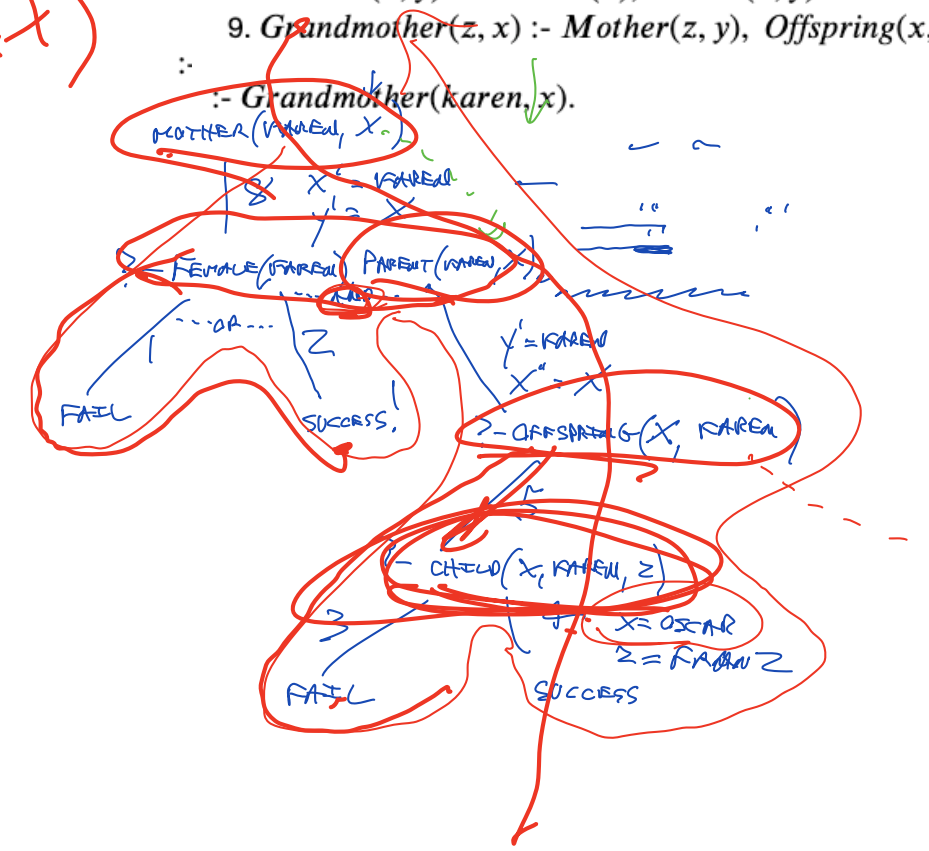
?- PARENT(KAREN, X)

?- OFFSPRING(X, KAREN)

?- CHILD(X, KAREN, Z)

YES

X = OSCAR



Prolog: Bells and Whistles....

A variety of other features make Prolog a realistic programming language:

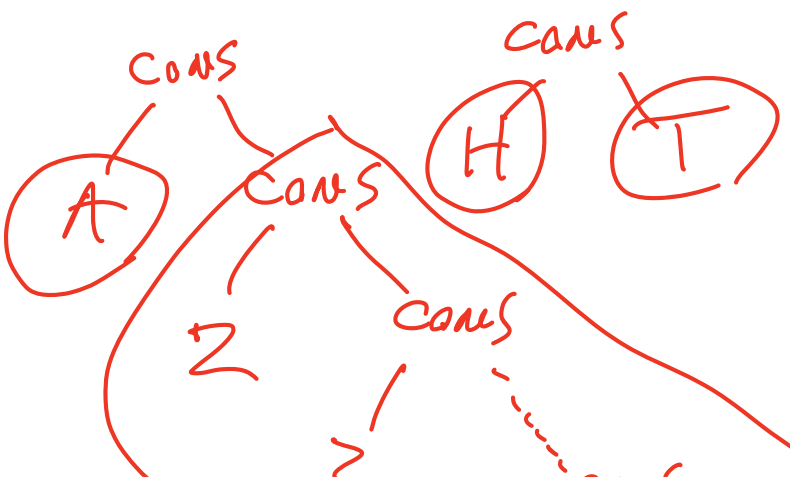
Arithmetic and Boolean operations can be included in programs:

```
1 max(X,Y,X) :- X >= Y.
2 max(X,Y,Y) :- X < Y.
```

Input and output primitives:

```
?- child_fact(X,Y,Z), write(X), write(' is a child of '),
write(Y), write(' and '), write(Z), write('.'), nl, fail.
```

Lists as a primary data structure:




```
list([A,2,2,B,3,4,5]).
```

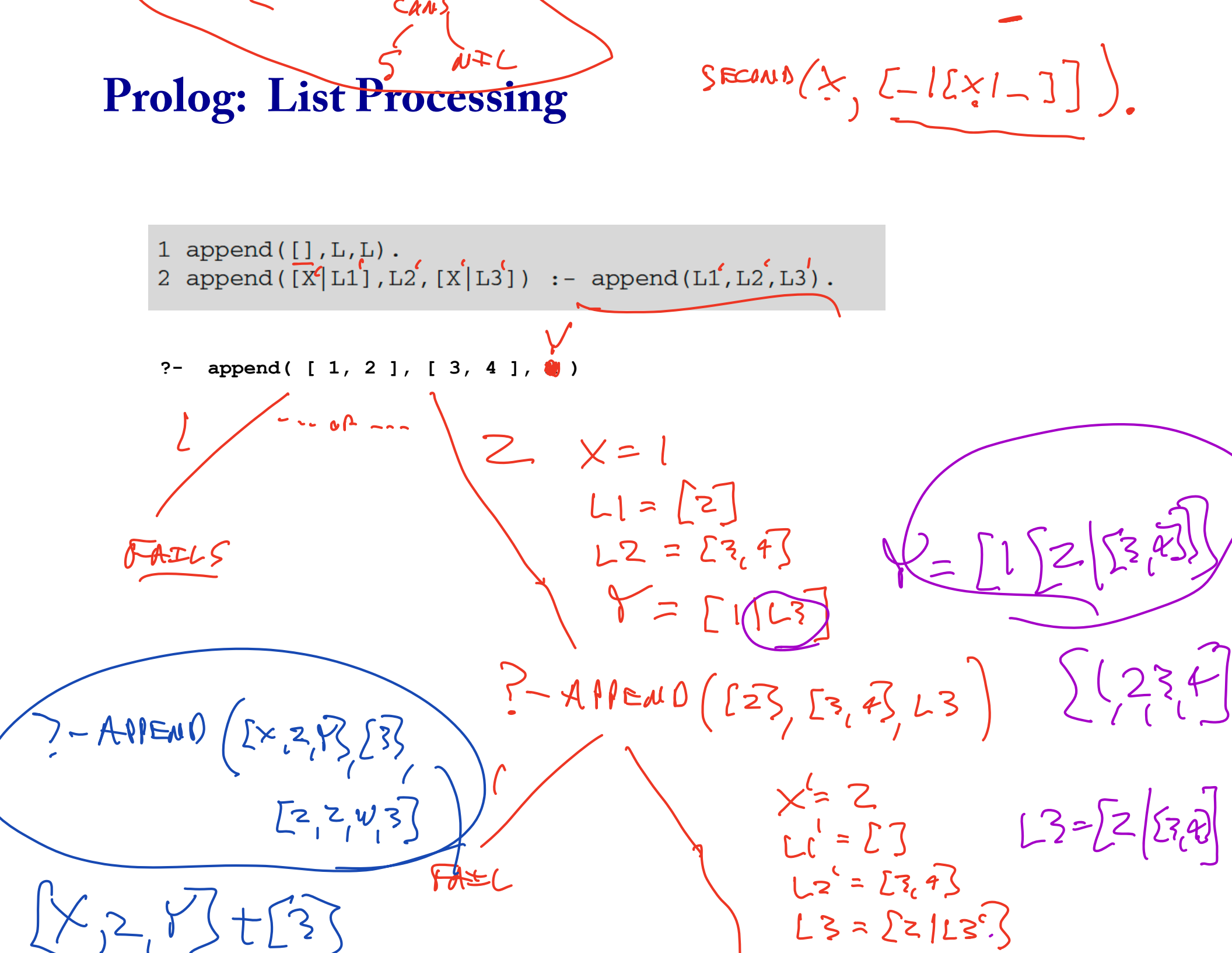
PROLOG displays the dialog

```
?- list([H|T]).

H = A
T = [2, 2, B, 3, 4, 5]

Yes
```

Prolog: List Processing

```
1 append([],L,L).
2 append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).
```

```
?- append( [ 1, 2 ], [ 3, 4 ], Y )
```



[x, z, w, 3]

x = 2, z = z, 8 = w

?- APPEND([3], [3,4], L3')

/

SUCCESS

L = [3,4]

L3' = [3,4]