

Programming Language Concepts and Semantics Part I

Lecture 3 The Truth About DCGs

To understand how DCGs translate into recursive descent parsers, we first must understand some PROLOG, the underlying Logic Programming Language. This Lecture will provide a quick introduction to Prolog

- The idea of declarative programming
- Logic rules as programs
- Execution of Prolog programs, backtracking
- Unification

The Roots: Database Programming

| <i>Children</i> | |
|-----------------|--------|
| Couple | Child |
| couple1 | roger |
| couple1 | lisa |
| couple2 | tom |
| couple2 | martin |

| <i>Marriage</i> | | |
|-----------------|----------|--------|
| Couple | Husband | Wife |
| couple1 | benjamin | stella |
| couple2 | scott | lisa |
| couple3 | tom | lydia |

In Prolog:

married(couple1, benjamin, stella).
married(couple2, lisa, scott).
married(couple3, tom, lydia).

child(couple1, roger).
child(couple1, lisa).
child(couple2, tom).
child(couple2, martin).

Queries:

“Who is Lisa married to?”

?- married(Couple, Husband, lisa).
-> Husband=scott,
Couple=couple2

“Who are the Children of Lisa and Scott?”

?- child(couple2, Child).
-> Child=tom
-> Child=martin

Rules in Deductive Databases

The Idea of deductive databases (and of Prolog) is to specify

- explicit data (as relations)

```
married(couple1, benjamin, stella).  
married(couple2, lisa, scott).  
married(couple3, tom, lydia).
```

```
child(couple1, roger).  
child(couple2, martin)...
```

- implicit data (via rules)

```
father(TheFather, Child) :-  
    married(Couple, TheFather, Mother),  
    child(Couple, Child).
```

but not (!) the way in which implicit data is derived.

Logic Programming

The dream of *Declarative Programming*

- don't program how to find solution
- model the problem and use a universal problem solving procedure to find the solution

Algorithm = Logic + Control

Procedural Interpretation of Rules

Logical deduction is the basis of programming

A if B1 and B2 and ... Bn

Is reinterpreted as

to *solve* A, first *solve* B1 then B2 then ... then Bn

or to *execute* A, first *execute* B1 then B2 then ... then Bn

Execution of Logic Programs

father(charles, philip).

father(ana, george).

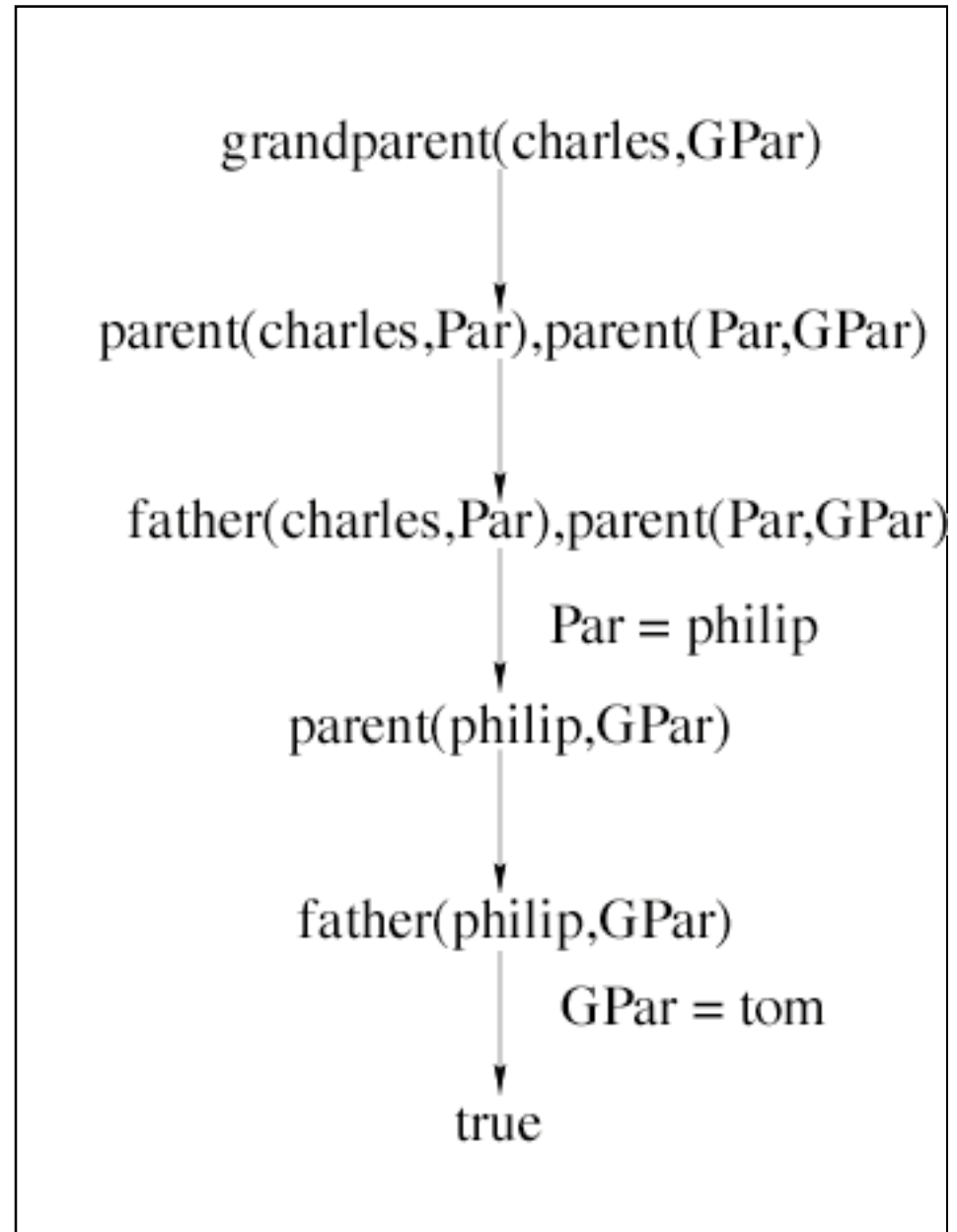
father(philip, tom).

mother(charles, ana).

parent(Pers, Par) :- father(Pers, Par).

parent(Pers, Par) :- mother(Pers, Par).

grandparent(Pers, Gpar) :-
parent(Pers, Par),
parent(Par, Gpar).



Simplified Interpreter (ground goals)

Input: A ground goal G and a program P

Output: yes if G is a consequence of P (“is true in P ”),
 no otherwise

Initialize resolvent to G

Algorithm:

While (resolvent A_1, \dots, A_n is not empty) do

 choose a goal A from the resolvent

 choose a ground instance of a clause

$A' :- B_1, \dots, B_n$ from P

 such that A and A' are identical

 (exit with “no” if no such clause)

 replace A by B_1, \dots, B_n in the resolvent

If (resolvent is empty) answer “yes” else answer “no”

Multiple Solutions

father(charles, philip).

father(ana, george).

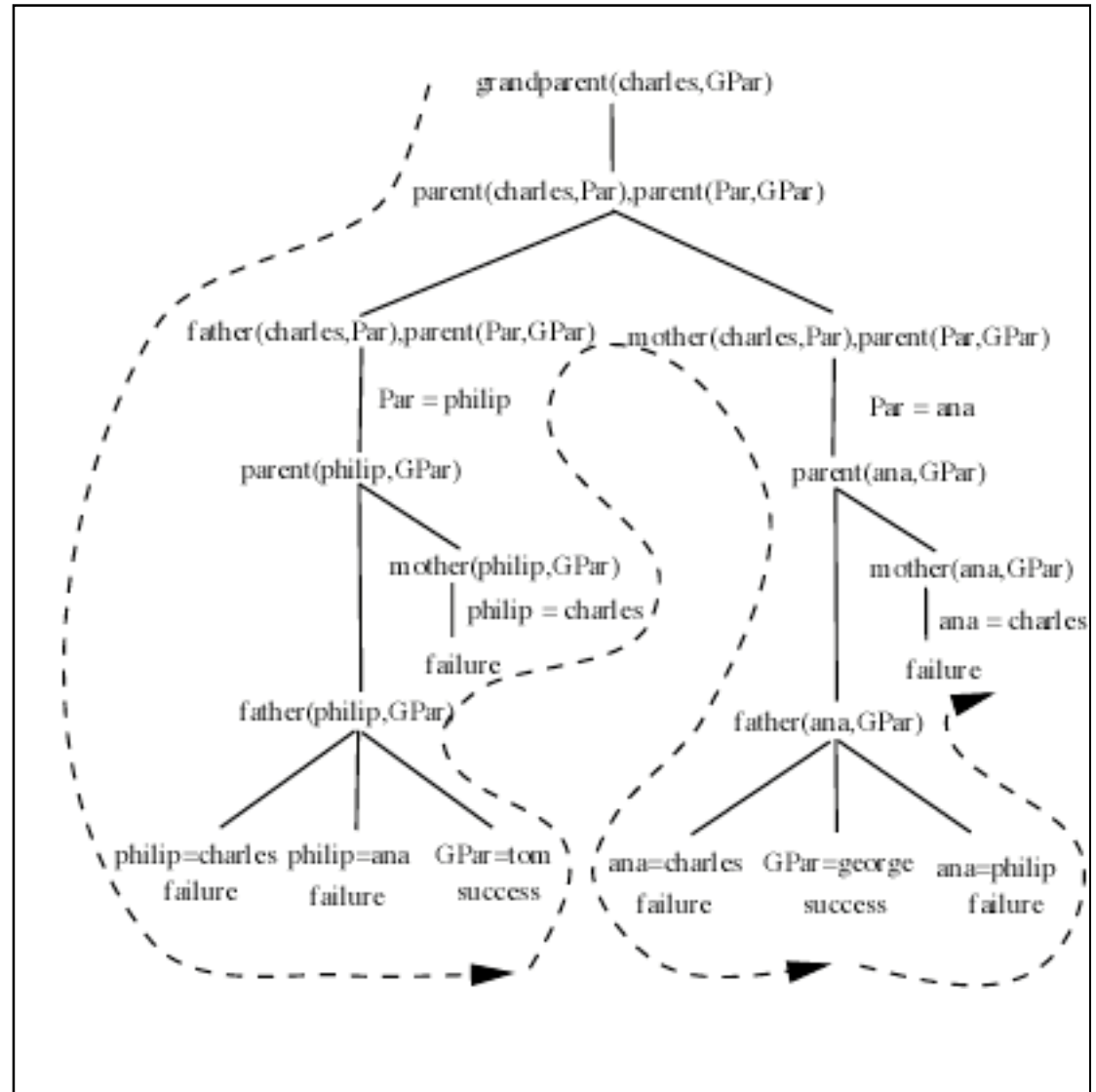
father(philip, tom).

mother(charles, ana).

parent(Pers, Par) :- father(Pers, Par).

parent(Pers, Par) :- mother(Pers, Par).

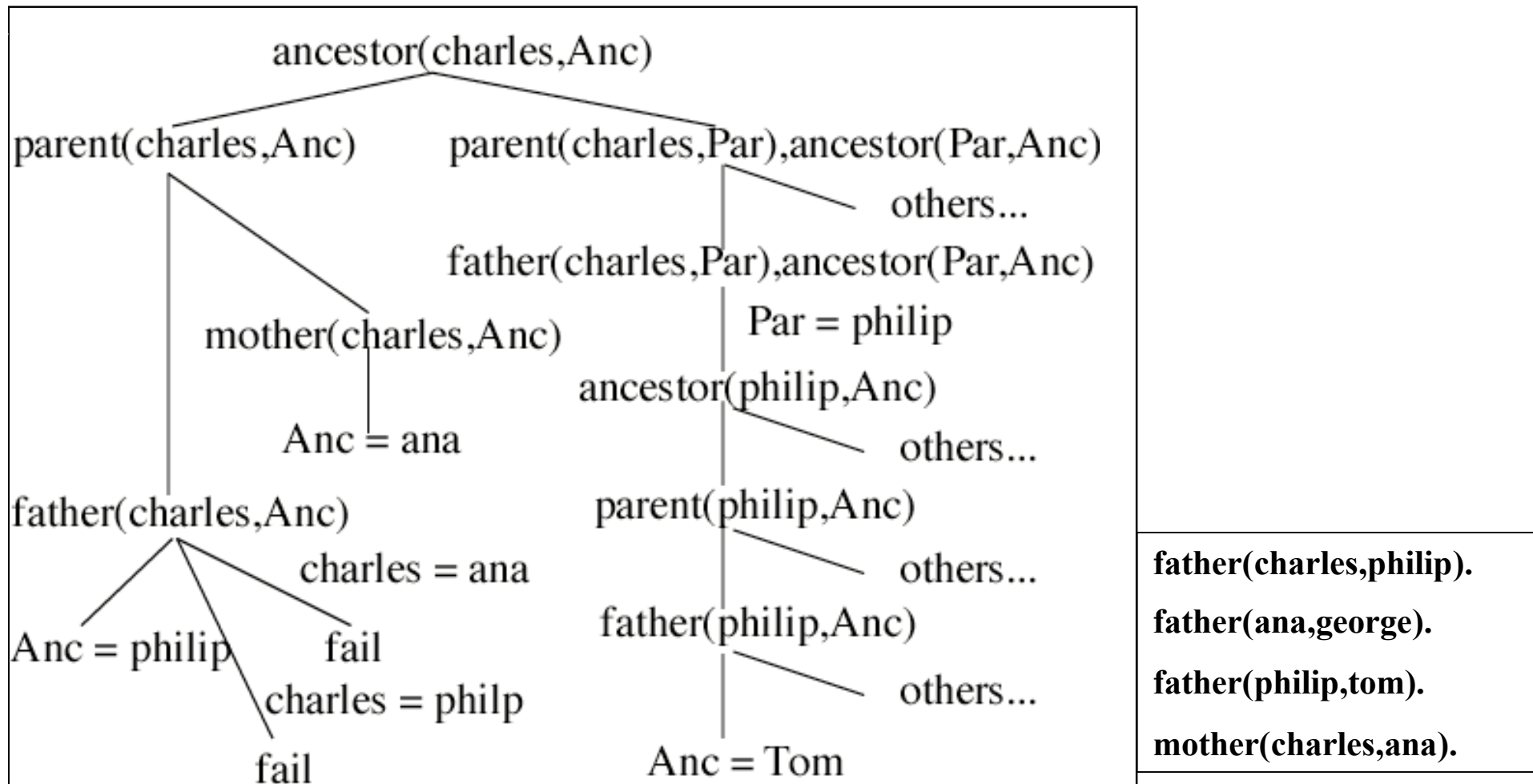
**grandparent(Pers, Gpar) :-
parent(Pers, Par),
parent(Par, Gpar).**



Recursion

ancestor(P, Anc) :- parent(P, Anc).

ancestor(P, Anc) :- parent(P, Par), ancestor(Par, Anc).



Data Structures

- ◆ CLP languages inherit data structures from Prolog:
 - ◆ Records
 - ◆ Lists
 - ◆ Trees
- ◆ Data structures are *terms*
- ◆ They are accessed and manipulated using *term constraints*
--they are just another constraint domain!
- ◆ There are no type declarations

Records

Struct person { becomes
 char[] Name;
 int age;
 char[] profession; }

person(Name, Age, Prof).

for example

person(martin, 36, lawyer).

Records can be nested:

person(Name, Age, Profession).

 profession(Type, Employer).

 e.g.

 person(john, 40,

 profession(pilot, quantas)).

Records

Records can be queried:

```
data(person(john, 40,  
           profession(pilot, quantas))).
```

“What is John’s profession?”

```
?- data(person(john, _Age, Prof)).
```

“Who is John’s employer?”

```
?- data(person(john, _Age, profession(_Type, Employer))).
```

“Who are the persons working as pilots?”

```
?- data(person(Name, _Age, profession(pilot, _Employer))).
```

Unification

- In (constraint) logic programming there are no assignments

$X = Y$ means

“ make X and Y equal, but keep them as general as possible”

$X=abc$

\Rightarrow

$X=abc$

$abc=X$

\Rightarrow

$X=abc$

$X=Y$

\Rightarrow

no change,

but both variables identical

(as if $X=_1$, $Y=_1$)

$f(a)=f(X)$

\Rightarrow

$X=a$

$f(a,Y)=f(X,X)$

\Rightarrow

$X=Y$ and $X=Y=a$

$f(X)=X$

\Rightarrow

??????? (not permitted). except in Prolog III)

Unification computes the substitution required to make the terms equal

Unification Algorithm (Idea)

Unify(C): C is a set of equations, S a substitution

Initialize S to empty

While C is not empty

 select equation c from C

 if c is of the form $X=X$ then remove c from C

 elseif c is of the form $f(X_1, \dots, X_n)=g(Y_1, \dots, Y_m)$ then

return false if not($f=g$) or not($m=n$)

 elseif c is of the form $f(X_1, \dots, X_n)=f(Y_1, \dots, Y_n)$ then

 replace c in C with $X_1=Y_1, \dots, X_n=Y_n$

 elseif c is of the form $\text{term}=X$ and X is a variable then

 replace c in C with $X=\text{term}$

 elseif c is of the form $X=\text{term}$ and X is a variable then

 if term contains X then return false

 else (1) remove c from C,

 (2) replace X with t throughout C and throughout S

 (3) add c to S

 endif

endif

endwhile

return S

Unification Exercise

• $abc = def$?

• $abc = Y$?

• $abc = f(Y)$?

• $f(abc) = f(Y)$?

• $g(abc) = f(Y)$?

• $g(abc, def, X) = g(A, Y, A)$?

• $g(abc, def, X) = g(A, A, A)$?

• $g(f(a), h(b)) = g(f(X), Y)$?

<- cf. record example

• $g(f(a), h(b)) = g(X, k(b))$?

• $X = g(X)$?

• $g(X, Y) = g(g(Y), a)$?

Lists

- ◆ Because lists are so important, special notation is used to represent both lists and partially constructed lists
- ◆ $[X_1, X_2, \dots, X_n]$ is a fixed size list containing the n elements X_1, X_2, \dots, X_n
- ◆ $[X_1, X_2, \dots, X_n | Y]$ is a list whose first n elements are X_1, X_2, \dots, X_n and whose remaining elements form the list Y .
- ◆ Special cases are $[],$ the empty or nil list and $[H | T]$ which is the list with head H and tail T .

Programming with Lists

- ◆ Logic languages do not provide iteration, only recursion.
- ◆ The key to programming with lists is to reason recursively about the list
 - ◆ Either the list is empty, and the operation of interest should be straightforward, or
 - ◆ It is non-empty and can be broken into a first element (the head) and the remaining list (the tail) which is dealt with recursively.

List Membership

- ◆ Define a predicate, `member(X,L)`, which holds if `X` is an element of list `L`.
- ◆ Key: reason about two cases for `L`
 - ◆ `L` is `[]`, *fail*
 - ◆ `L` is `[H|T]`:
 - ◆ if `H` is `X` then true, or
 - ◆ if `X` is an element of `T` it is true.

~~`member(X,[]) :- fail.`~~

~~`member(X, [H|T]) :- X=H.`~~

~~`member(X, [H|T]) :- member(X,T).`~~

`member(X, [X|_]).`

`member(X, [_|T]) :- member(X,T).`

Membership Examples

`member(X, [X|_]).`

`member(X, [_|T]) :- member(X,T).`

- Is X a member of L ?

`?- member(1,[2,3,1,4]).`

Yes

- What are the elements in L ?

`?- member(X,[2,3,1,4]).`

$X=2$; $X=3$; $X=1$; $X=4$; no (more answers)

- What lists contain X as an element?

`?- member(X,L).`

$L=[X|_]$; $L=[_,X|_]$; $L=[_,_,X|_]$; ...

DCGs

There is a straight forward translation of DCGs into Prolog Programs.

This is (in principle) the translation that Prolog performs when it consults a DCG.

```
factor --> 'id' .
```

```
factor(Input, Rest) :- Input = ['id' | Rest].
```

```
factor --> ['(', exp, [')'].
```

```
factor(Input, Rest) :- Input = ['(' | Rest1],  
                        exp(Rest1, Rest2), Rest2=[')']|Rest].
```

```
exp --> term, exp1.
```

```
exp(Input, Rest) :- term(Input, Rest1), exp1(Rest1, Rest).
```

You can see how each non-terminal translates into a predicate with two arguments: the first argument is the input sequence, the second argument returns the remainder of the input after removing the phrase that constitutes this non-terminal from the beginning of the input.

- Terminal symbols are simply removed by unification
- Non-Terminal call each other
- Calls on the RHS are chained through appropriate auxiliary variables.

Summary

We have looked at Prolog's execution mechanism

- Prolog is a declarative language based on predicate logic
- Prolog supports search / backtracking directly
- Program execution in Prolog corresponds to an automatic proof
- Grammars have a natural translation in Prolog
- Thus recursive descent parsing can be understood as a proof (that the given phrase is of a particular grammatical structure).

Homework

How does a DCG execute in Prolog? Use the DCG for extended expressions you wrote in the last homework and translate it into manually into a Prolog program. Use this program to parse expressions.

Now read in the your DCG directly using consult and use "listing." to see its translation.

What do you observe? Give an appropriate definition of 'C'(X, Y, Z)?

Appendix: Syntax

Variable

- sequence of alphanumeric characters + “_”
- starting with uppercase alphabetic or underscore

Valid: X, Name, BlaBla, Test3, _, _1, _lowercase

Invalid: x, 1X, testing, :bla, \$notavar

Constant

- sequence of alphanumeric characters + “_”
 - starting with lowercase alphabetic
- numeric sequence
- any sequence of characters in single quotes
- some special characters (&+-. /:;<=>?)

Valid: x, x1, x_1, test7, ‘this is a const’, 8992

Invalid: NotAConst, “kdhkjf”, _22, 8...9

Rules and Predicates (the equivalent of procedures)

- *Atom*: $p(T_1, \dots, T_n)$

p is an n -ary predicate symbol (constant syntax)

$T_1 \dots T_n$ are terms (variable and constants, more forms later)

- *Rule / Clause*: Head :- Body

$\text{head}(T_1, \dots, T_k) \text{ :- } \text{atom}_1(S_{1,1}, \dots, S_{1,n_1}), \dots, \text{atom}_k(S_{k,1}, S_{k,n_k}).$

Example: $\text{mother}(M,C) \text{ :- } \text{child}(X,C), \text{married}(X,P,M).$
a rule describes "one case" of a predicate

- Logic meaning: head is true if body is true
- Procedural meaning:
to solve head,
solve $\text{atom}_1(S_{1,1}, \dots, S_{1,n_1})$, then ... then $\text{atom}_k(S_{k,1}, S_{k,n_k}).$

Rules and Predicates (cont'd)

- *Fact*: person(joe).

a predicate with an empty body.

- *Predicate definition*:

a set of rules for the same predicate symbol.
Interpreted as a disjunction.

```
parent(Person, Parent) :- father(Person, Parent).  
parent(Person, Parent) :- mother(Person, Parent).
```

- *Query*: syntactically just a body

normally with free variables that are bound in the answer.

```
parent(martin, X). -> the answer is X=scott, X=lisa.
```