XSB Prolog

CSE 392, Computers Playing Jeopardy!, Fall 2011

Stony Brook University

http://www.cs.stonybrook.edu/~cse392

- IBM Watson Question Analysis for Jeopardy! = UIMA + Prolog

What Is Prolog?

- Prolog is a logic-based language
- Simple Knowledge Representation
- With a few simple rules, information can be analyzed
  - Socrates is a man.
  - All men are mortal.
  - Therefore, Socrates is mortal.
- This is logic. Can Prolog do it?
  - Yes, but infinite in some cases
- XSB = Prolog + tabling
  - better termination properties

Brief History

- The first, official version of Prolog was developed
  - at the University of Marseilles, France by Alain Colmerauer in the early 1970s
  - as a tool for PROgramming in LOGic.

Application Areas

- Prolog has been a very important tool in
  - artificial intelligence applications
  - expert systems
  - natural language interfaces
  - smart information management systems

Declarative Language

- This means that
  - The programmer
    - declares **facts**
    - defines **rules** for reasoning with the facts
  - Prolog uses deductive reasoning to
    - decide whether a proposed fact (**goal**) can be logically derived from known facts

      (such a decision is called a **conclusion**)
    - determine new facts from old

Monotonic logic

- Standard logic is monotonic: once you prove something is true, it is true forever
- Logic isn't a good fit to reality
- NOT = negation as failure
  - illegal(X) :- \+ legal(X).
  - If no proof can be found, the original goal succeeds.

Nonmonotonic logic

- A non-monotonic logic is a formal logic whose consequence relation is not monotonic.
- Adding a formula to a theory produces a reduction of its set of consequences.

  `p:- \+ q.`

  - p is true because q is not **known/derivable** to be true
  - What if later q is **asserted**? Then p is false.
  - The \+/1 prefix operator is called the "not provable" operator, since the query ?- \+ Goal. succeeds if Goal is not provable.
- XSB Prolog uses nonmonotonic logic

Formalizing Arguments

- Abstracting with symbols for predicates, we get an argument form that looks like this:

  if p then q
  p
  therefore q

- $((q :- p) \wedge p) \Rightarrow q$

Forward and backward reasoning

- A syllogism gives two premises, then asks, "What can we conclude?"
  - This is **forward reasoning** -- from premises to conclusions
  - it's inefficient when you have lots of premises
- Instead, you ask Prolog specific questions
  - Prolog seeks for the goals provided by the user as questions
  - Prolog uses backward reasoning -- from (potential) conclusions to facts
    - Prolog searches successful paths and if it reaches unsuccessful branch, it backtracks to previous one and tries to apply alternative clauses

Prolog: Facts, Rules and Queries

	Prolog
Socrates is a man.	`man(socrates).`
All men are mortal.	`mortal(X) :- man(X).`
Is Socrates mortal?	`?- mortal(socrates).`

Facts, rules, and queries

- Fact: Socrates is a man.

```
man(socrates).
```

- Rule: All men are mortal.

```
mortal(X) :- man(X).
```

- Query: Is Socrates mortal?

```
?-  mortal(socrates).
```

Running XSB Prolog

- Install XSB Prolog
  - Windows distribution
  - build/configure and make for Linux and MacOS
- Create your "database" (program) in any editor

```
man(socrates).
mortal(X) :- man(X).
```

- Save it as *text only*, with a **.P** extension (or .pl)
- Run xsb

```
?- consult('socrates.pl').
```

- Then, ask your question at the prompt:

```
?- mortal(socrates).
```

Prolog is a theorem prover

- Prolog's "Yes" means "I can prove it“
- Prolog's "No" means "I can't prove it"

```
?- mortal(plato).
   No
```

- XSB Prolog has closed world assumption: knows everything it needs to know
- Prolog supplies values for variables when it can
  - ?- mortal(X).
    X = socrates

Prolog Example: Reachability

```
edge(1,2).
edge(2,3).
edge(2,4).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z, Y).
```

Prolog Example: Reachability

```
  | ?- reachable(X,Y).

X = 1
Y = 2;
```

Type a semi-colon repeatedly

```
X = 2
Y = 3;

X = 2
Y = 4;

X = 1
Y = 3;

X = 1
Y = 4;

no

| ?- halt.
```

Command to Exit XSB

XSB Prolog Example: Reachability

```
edge(1,2).
edge(2,3).
edge(2,4).
edge(4,1).
:- table(reachable/2).
reachable(X,Y) :- edge(X,Y).
reachable(X,Y) :- edge(X,Z), reachable(Z, Y).
```

Prolog

- A *predicate* is a collection of clauses with the same *functor* (name) and *arity* (number of arguments).

```
parent(paul,steven).
parent(peter,olivia).
parent(tom,liz).
parent(tony, ann).
parent(michael,paul).
parent(jill,tania).
```

- A *program* is a collection of predicates.
- Clauses within a predicate are used in the order in which they occur.

Syntax

- Variables begin with a capital letter or underscore:

  `X, Socrates, _result`

- Atoms do *not* begin with a capital letter:

  `socrates, paul`

  - Atoms containing special characters, or beginning with a capital letter, must be enclosed in single quotes: `‘Socrates’`

Data types

- An **atom** is a general-purpose name with no inherent meaning.
- **Numbers** can be floats or integers.
- A **compound term** is composed of an atom called a "functor" and a number of "arguments", which are again terms: tree(node(a),tree(node(b),node(c)))
- Special cases of compound terms:
  - *Lists:* ordered collections of terms: [], [1,2,3], [a,1,X|T]
  - *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes: “abc”, “to be, or not to be”

Representation of Lists

- List is handled as binary tree in Prolog

  [Head | Tail]   OR

  .(Head, Tail)

  - Where Head is an atom and Tail is a list
  - We can write [a,b,c] or .(a,.(b,.(c,[]))).

Matching

- Given two terms, they are identical or the variables in both terms can have same objects after being instantiated

  date(D,M,2006) unification date(D1,feb,Y1)

  D=D1, M=feb, Y1=2006

- General Rule to decide whether two terms, S and T match are as follows:
  - If S and T are constants, S=T if both are same object
  - If S is a variable and T is anything, T=S
  - If T is variable and S is anything, S=T
  - If S and T are structures, S=T if
    - S and T have same functor
    - All their corresponding arguments components have to match

Prolog Evaluation

- Execution of a Prolog program is initiated by the user's posting of a single goal, called the **query**.
  - SLD resolution
    - If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program.

Declarative and Procedural Way

- Prolog programs can be understood two ways: declaratively and procedurally.
- P:- Q,R
- Declarative Way
  - P is true if Q and R are true
- Procedural Way
  - To solve problem P, first solve Q and then R (or) To satisfy P, first satisfy Q and then R
  - Procedural way does not only define logical relation between the head of the clause and the goals in the body, but also the order in which the goal are processed.

Formal Declarative Meaning

- Given a program and a goal G,
- A goal G is true (that is satisfiable, or logically follows from the program) if and only if:
  - There is a clause C in the program such that
  - There is a clause instance I of C such that
    - The head of I is identical to G, and
    - All the goals in the body of I are true.

Evaluation

```
mother_child(trude, sally).

father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).

sibling(X, Y):- parent_child(Z, X), parent_child(Z, Y).

?- sibling(sally, erica).
Yes
```

(by chronological backtracking)

Evaluation

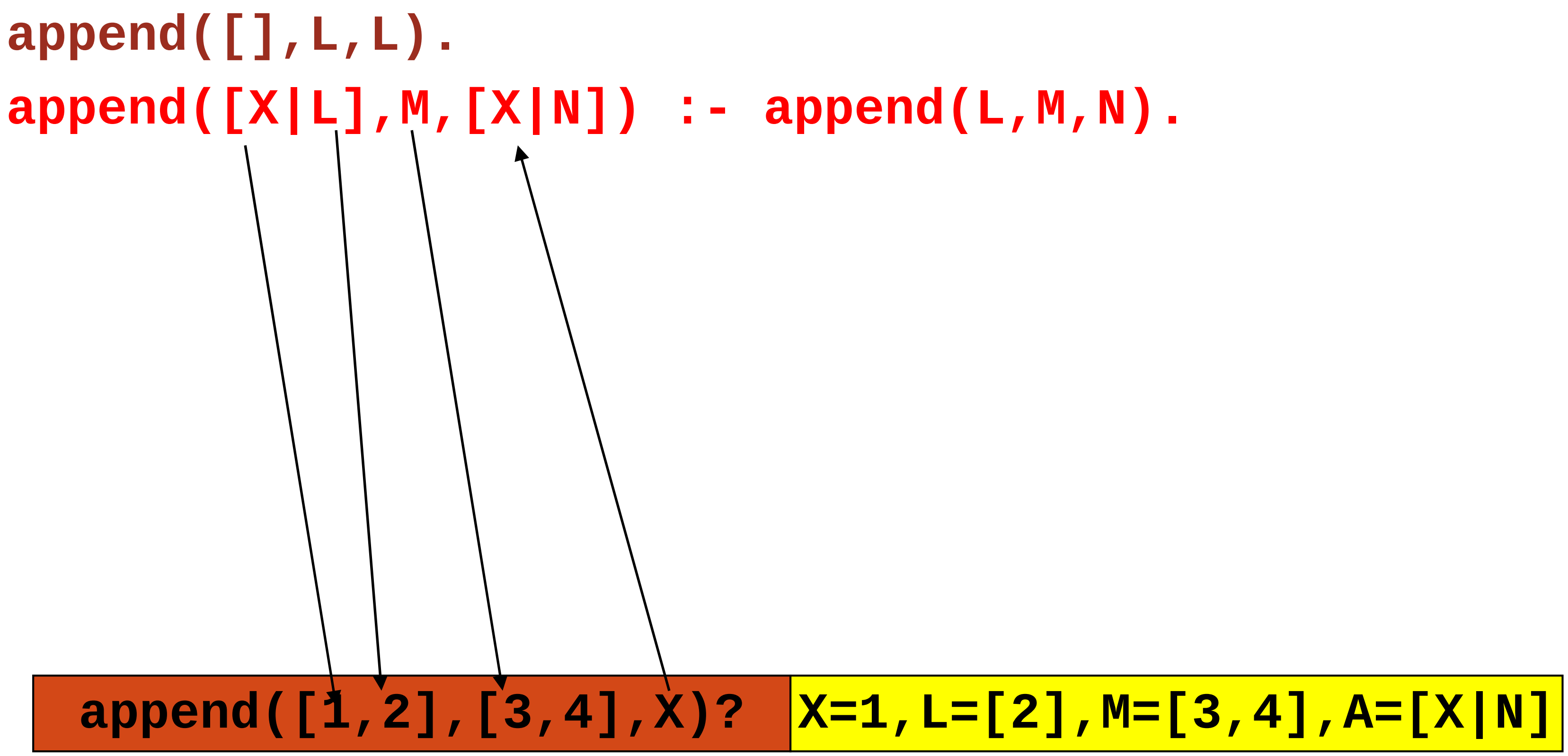
- **`?- father_child(Father, Child).`**
  enumerates all valid answers on backtracking.

Append example

```
append([],L,L).
append([X|L], M, [X|N]) :- append(L,M,N).
```

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

Append example



```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

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

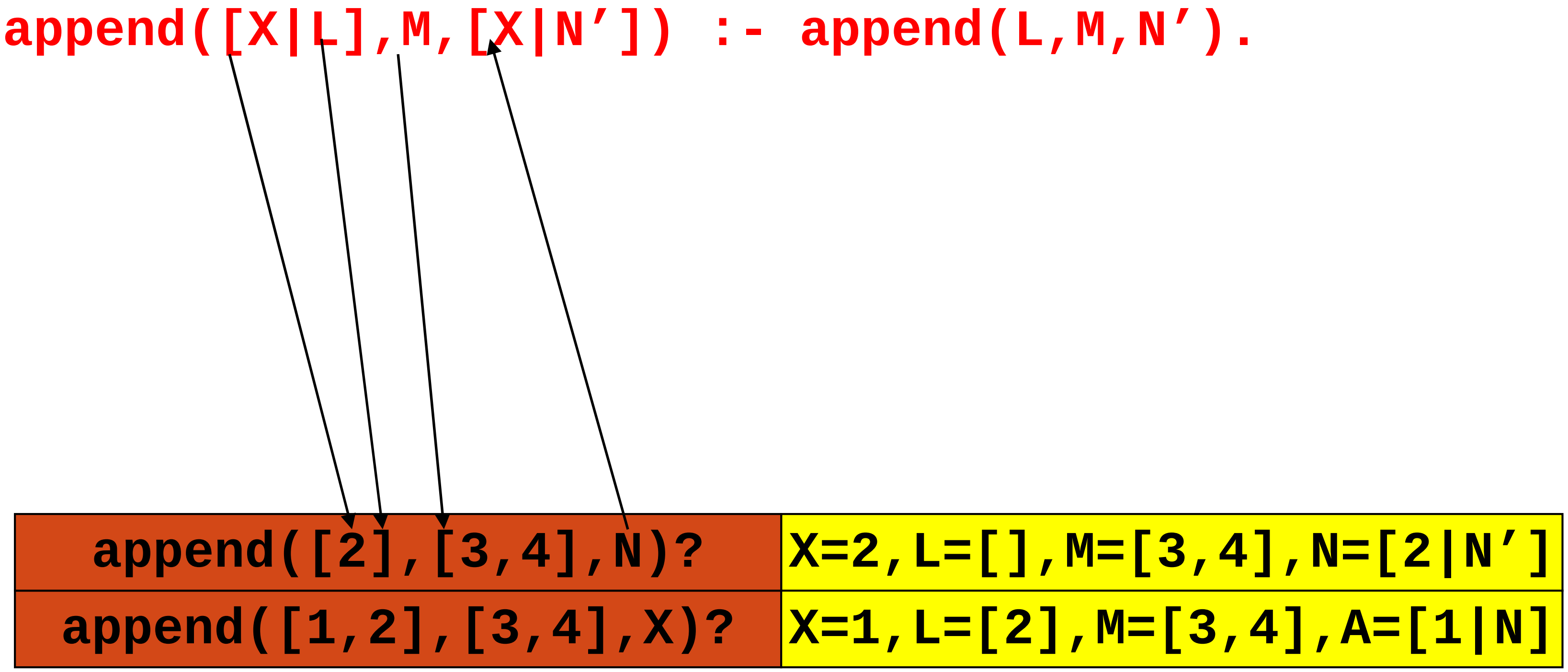
```
X=1,L=[2],M=[3,4],A=[X|N]
```

Append example

```
append([],L,L).
append([X|L],M,[X|N]) :- append(L,M,N).
```

```
append([2],[3,4],N)?
append([1,2],[3,4],X)?  X=1,L=[2],M=[3,4],A=[X|N]
```

Append example



```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

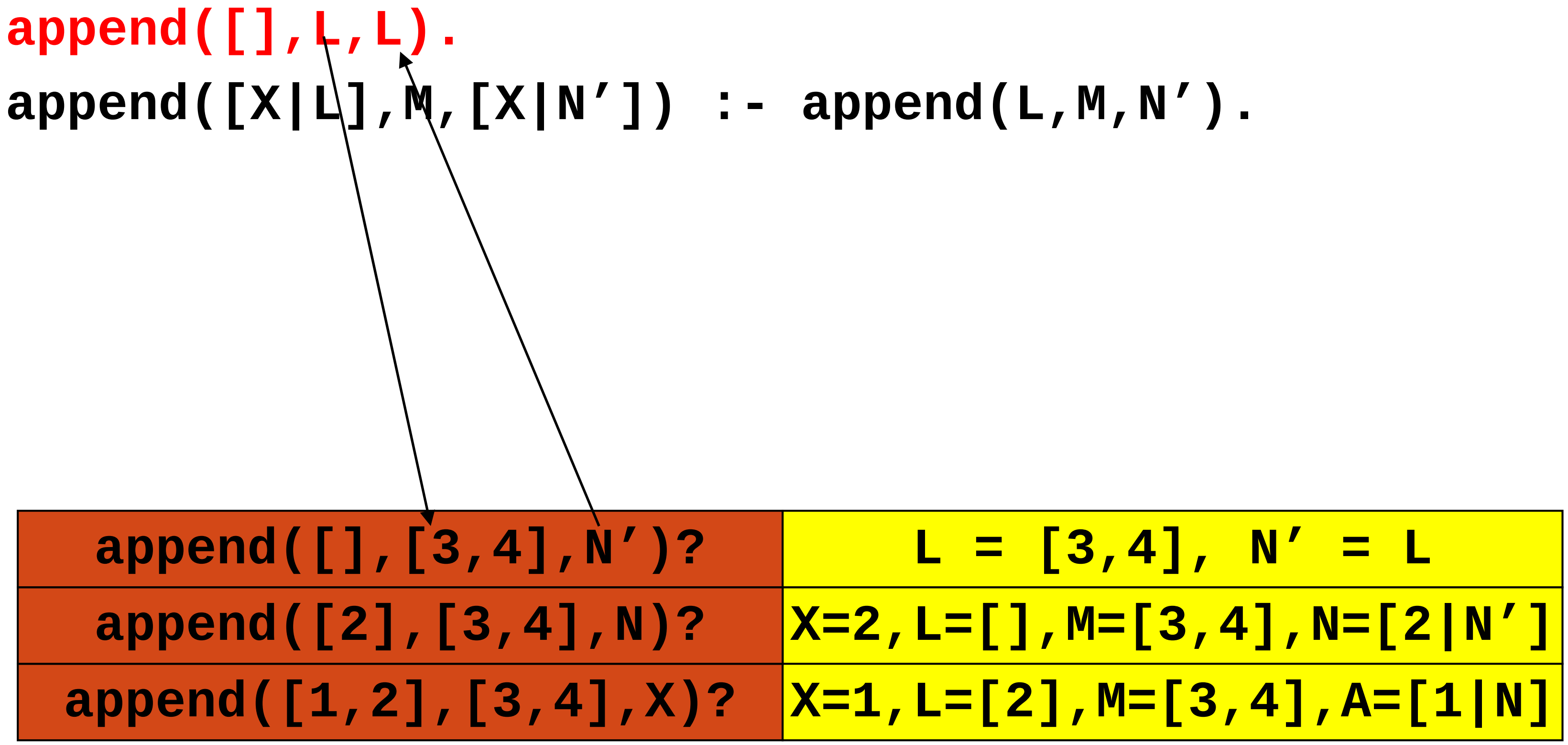
append([2],[3,4],N)?	**X=2,L=[],M=[3,4],N=[2|N’]**
append([1,2],[3,4],X)?	**X=1,L=[2],M=[3,4],A=[1|N]**

Append example

```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

append([],[3,4],N’)?	
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N’]
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example



```
append([],L,L).
append([X|L],M,[X|N′]) :- append(L,M,N′).
```

append([],[3,4],N′)?	L = [3,4], N′ = L
append([2],[3,4],N)?	X=2,L=[],M=[3,4],N=[2|N′]
append([1,2],[3,4],X)?	X=1,L=[2],M=[3,4],A=[1|N]

Append example

```
append([],L,L).
append([X|L],M,[X|N’]) :- append(L,M,N’).
```

```
A = [1|N]
 N = [2|N’]
 N’= L
 L = [3,4]
```

Answer: **A** = **[1,2,3,4]**

`append([],[3,4],N’)?`	`L = [3,4], N’ = L`
`append([2],[3,4],N)?`	`X=2,L=[],M=[3,4],N=[2|N’]`
`append([1,2],[3,4],X)?`	`X=1,L=[2],M=[3,4],A=[1|N]`

Quicksort Example

```
partition([], _, [], []).
partition([X|Xs], Pivot, Smalls, Bigs) :-
    (   X @< Pivot ->
        Smalls = [X|Rest],
        partition(Xs, Pivot, Rest, Bigs)
    ;   Bigs = [X|Rest],
        partition(Xs, Pivot, Smalls, Rest)
    ).
quicksort([])     --> [].
quicksort([X|Xs]) -->
    { partition(Xs, X, Smaller, Bigger) },
    quicksort(Smaller), [X], quicksort(Bigger).
```

Interfaces to Java

- XSB Prolog: InterProlog (native | | sockets)
- SWI-Prolog: JPL (native)
- Sicstus: PrologBeans (sockets)

More Examples

```
member(X,[X|R]).
member(X,[Y|R]) :- member(X,R)
```

- *X is a member of a list whose first element is X.*
- *X is a member of a list whose tail is R if X is a member of R.*

```
?- member(2,[1,2,3]).
Yes
?- member(X,[1,2,3]).
X = 1 ;
X = 2 ;
X = 3 ;
No
```

More Examples

```
select(X,[X|R],R).
select(X,[F|R],[F|S]) :- select(X,R,S).
```

- *When X is selected from [X|R], R results.*
- *When X is selected from the tail of [X|R], [X|S] results, where S is the result of taking X out of R.*

```
?- select(X,[1,2,3],L).
X=1  L=[2,3] ;
X=2  L=[1,3] ;
X=3  L=[1,2] ;
No
```

More Examples

```
append([],X,X).
append([X|Y],Z,[X|W]) :- append(Y,Z,W).

?- append([1,2,3],[4,5],X).
X=[1,2,3,4,5]
Yes
```

More Examples

```
reverse([X|Y],Z,W) :- reverse(Y,[X|Z],W).
reverse([],X,X).

?- reverse([1,2,3],[],X).
X = [3,2,1]
Yes
```

More Examples

```
perm([],[]).
perm([X|Y],Z) :- perm(Y,W), select(X,Z,W).

?- perm([1,2,3],P).
P = [1,2,3] ;
P = [2,1,3] ;
P = [2,3,1] ;
P = [1,3,2] ;
P = [3,1,2] ;
P = [3,2,1]
```

More Examples

- Sets

```
union([X|Y],Z,W) :- member(X,Z),  union(Y,Z,W).
union([X|Y],Z,[X|W]) :- \+ member(X,Z), union(Y,Z,W).
union([],Z,Z).

intersection([X|Y],M,[X|Z]) :- member(X,M), intersection(Y,M,Z).
intersection([X|Y],M,Z) :- \+ member(X,M), intersection(Y,M,Z).
intersection([],M,[]).
```

Definite clause grammar (DCG)

- A **DCG** is a way of expressing grammar in a logic programming language such as Prolog
- The definite clauses of a DCG can be considered a set of axioms where the fact that it has a parse tree can be considered theorems that follow from these axioms

DCG Example

```
sentence --> noun_phrase, verb_phrase.
noun_phrase --> det, noun.
verb_phrase --> verb, noun_phrase.
det --> [the].
det --> [a].
noun --> [cat].
noun --> [bat].
verb --> [eats].

?- sentence(X,[]).
```

DCG

- Not only context-free grammars
- Context-sensitive grammars can also be expressed with DCGs, by providing extra arguments

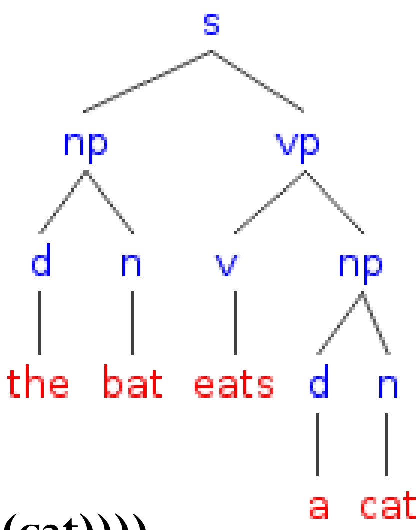
```
s --> symbols(Sem,a), symbols(Sem,b), symbols(Sem,c).
symbols(end,_) --> [].
symbols(s(Sem),S) --> [S], symbols(Sem,S).
```

DCG

```
sentence --> pronoun(subject), verb_phrase.
verb_phrase --> verb, pronoun(object).
pronoun(subject) --> [he].
pronoun(subject) --> [she].
pronoun(object) --> [him].
pronoun(object) --> [her].
verb --> [likes].
```

Parsing with DCGs

```
sentence(s(NP,VP)) --> noun_phrase(NP), verb_phrase(VP).
noun_phrase(np(D,N)) --> det(D), noun(N).
verb_phrase(vp(V,NP)) --> verb(V), noun_phrase(NP).
det(d(the)) --> [the].
det(d(a)) --> [a].
noun(n(bat)) --> [bat].
noun(n(cat)) --> [cat].
verb(v(eats)) --> [eats].
```



```
?- sentence(Parse_tree, [the,bat,eats,a,cat], []).
Parse_tree = s(np(d(the),n(bat)),vp(v(eats),np(d(a),n(cat))))
```

```
s --> np, vp.
np --> det, n.
vp --> tv, np.
vp --> v.
det --> [the].
det --> [a].
det --> [every].
n --> [man].
n --> [woman].
n --> [park].
tv --> [loves].
tv --> [likes].
v --> [walks].
```

```
| ?- s([a,man,loves,the,woman],[]).
yes
| ?- s([every,woman,walks],[]).
yes
| ?- s([a,woman,likes,the,park],[]).
yes
| ?- s([a,woman,likes,the,prak],[]).
no
```

Cut (logic programming)

- Cut (! in Prolog) is a goal which always succeeds, but cannot be backtracked past
- **Green cut**

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X), \+ gotmoney(X).
```

  - **cut says "stop looking for alternatives"**
  - by explicitly writing \+ gotmoney(X), it guarantees that the second rule will always work even if the first one is removed by accident or changed
- **Red cut**

```
gamble(X) :- gotmoney(X),!.
gamble(X) :- gotcredit(X).
```