More on implementing finite state machines in PROLOG

- Recursive relations in PROLOG:
  - data structures needed
  - two example relations
- Completing the FSM recognition and generation algorithms to use
  - ϵ transitions
  - abbreviations

Recursive relations in PROLOG
Compound terms as data structures

To define recursive relations, one needs a richer data structure than the constants (atoms) introduced so far: *compound terms*.

A compound term comprises a functor and a sequence of one or more terms, the argument.[1] Compound terms are standardly written in prefix notation.[2]

For example:

- `bin_tree(`*mother*`, `*l-dtr*`, `*r-dtr*`)`
- `bin_tree(s, np, bin_tree(vp,v,n))`

[1] An atom can be thought of as a functor with arity 0.

[2] Infix and postfix operators can also be defined, but need to be declared.

Recursive relations in PROLOG
Lists as special compound terms

Lists are represented as compound terms.

- symbol "." as binary functor
- first argument: first element of list
- second argument: rest of list
- empty list: represented by the atom "[]"

Example: `.(a, .(b, .(c, .(d,[]))))`

Abbreviatory syntax available:

- bracket notation: `[` *`element1`* `|` *`restlist`* `]`

  Example: `[a | [b | [c | [d | []]]]]`

- element separator: `[` *`element1`* `,` *`element2`*`]`
  $=$ `[` *`element1`* `| [`*`element2`* `| []]]`

  Example: `[a, b, c, d]`

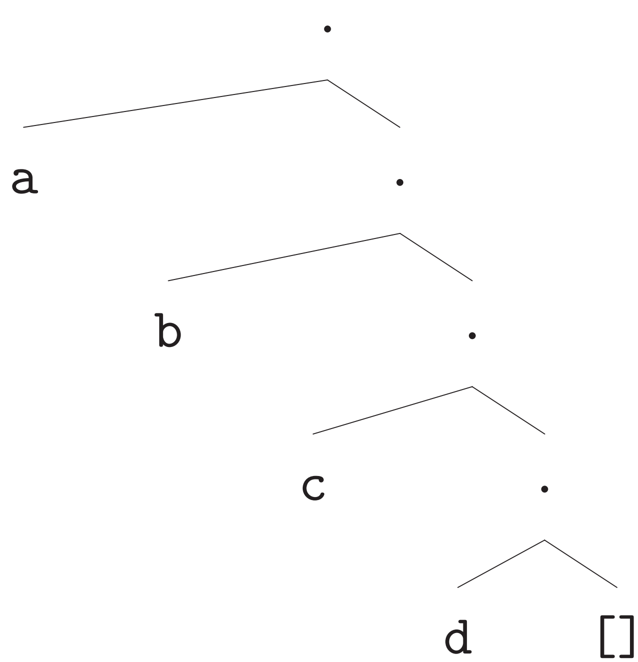
Four equivalent representations for lists:

1. `[a,b,c,d]`

2. `[a | [b | [c | [d | []]]]]`

3. `.(a, .(b, .(c, .(d,[]))))`

4.



Recursive relations in PROLOG
Example relations I: append

- Idea: a relation concatenating two lists
- Example:
  `?- append([a,b,c],[d,e],X).`
  $\Rightarrow$ `X=[a,b,c,d,e]`

```
append([],L,L).
append([H|T],L,[H|R]) :-
        append(T,L,R).
```

Recursive relations in PROLOG

Example relations II: reverse

- Idea: reverse a list
- Example: `?- reverse([a,b,c],X).` $\Rightarrow$ `X=[c,b,a]`

1. naive reverse:

```
naive_reverse([],[]).
naive_reverse([H|T],Result) :-
        naive_reverse(T,Aux),
        append(Aux,[H],Result).
```

2. reverse:

```
reverse(A,B) :-
        reverse_aux(A,[],B).

reverse_aux([],L,L).
reverse_aux([H|T],L,Result) :-
        reverse_aux(T,[H|L],Result).
```

Negation in PROLOG

- PROLOG does not have the means to express `not(P)` in the sense that `P` is known to be false.

- Instead, PROLOG has so-called *negation by failure*. Negating a goal P in PROLOG means that the system will try to prove P and if that fails, not(P) will be true.

- As notation for negation, the unary operator \+ is used. To use the functor `not` instead, one can simply define: `not(X) :- \+(X).`

FSMs with ϵ transitions and abbreviations
Defining PROLOG representations

1. Decide on a symbol to use to mark ϵ transitions: `'#'`

2. Define abbreviations for labels: `macro(Label,Word).`

3. Define a relation `special/1` to recognize abbreviations and epsilon transitions:

```
special(#).
special(X) :-
        macro(X,_).
```

FSMs with ϵ transitions and abbreviations
Extending the recognition algorithm

```
test(Words) :-
        initial(Node),
        recognize(Node,Words).

recognize(Node,[]) :-
        final(Node).
recognize(FromNode,String) :-
        arc(FromNode,Label,ToNode),
        traverse(Label,String,NewString),
        recognize(ToNode,NewString).

traverse(Label,[Label|RestString],RestString) :-
    not(special(Label)).
traverse(Abbrev,[Label|RestString],RestString) :-
    macro(Abbrev,Label).
traverse('#',String,String).

special(#).
special(X) :-
        macro(X,_).

not(X) :- \+(X).
```