Implementing finite state machines and learning Prolog along the way

Detmar Meurers: Intro to Computational Linguistics I
OSU, LING 684.01

Overview

- A first introduction to Prolog
- Encoding finite state machines in Prolog
- Recognition and generation with finite state machines in Prolog
- Completing the FSM recognition and generation algorithms to use
  - ϵ transitions
  - abbreviations
- Encoding finite state transducers in Prolog

The Prolog programming language (1)

PROgrammation LOGique was invented by Alain Colmerauer and colleagues at Marseille and Edinburgh in the early 70s. A Prolog program is written in a subset of first order predicate logic. There are

- **constants** naming entities
  - *syntax*: starting with lower-case letter (or number or single quoted)
  - *examples:* `twelve, a, q_1, 14, 'John'`
- **variables** over entities
  - *syntax*: starting with upper-case letter (or an underscore)
  - *examples:* `A, This, _twelve, _`
- **predicate symbols** naming relations among entities
  - *syntax:* predicate name starting with a lower-case letter with parentheses around comma-separated arguments
  - *examples:* `father(tom,mary)`, `age(X,15)`

The Prolog programming language (2)

A Prolog program consists of a set of *Horn* clauses:

- **unit clauses** or **facts**
  - *syntax:* predicate followed by a dot
  - *example:* `father(tom,mary).`
- **non-unit clauses** or **rules**
  - *syntax:* rel_0 `:-` rel_1`, ...,` rel_n`.`
  - *example:*

```
grandfather(Old,Young) :-
      father(Old,Middle),
      father(Middle,Young).
```


The Prolog programming language (3)

- No global variables: Variables only have scope over a single clause.
- No explicit typing of variables or of the arguments of predicates.
- Negation by failure: For `\+(P)` Prolog attempts to prove P, and if this succeeds, it fails.

A first Prolog program
grandfather.pl

```
father(adam,ben).
father(ben,claire).
father(ben,chris).

grandfather(Old,Young) :-
   father(Old,Middle),
   father(Middle,Young).
```

Query:

```
?- grandfather(adam,X).
X = claire ? ;
X = chris ? ;
no
```


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]

Example:

- binary tree: `bin_tree(`*mother*`,` *l-dtr*`,` *r-dtr*`)`
- example: `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

- empty list: represented by the atom "`[]`"
- non-empty list: compound term with "`.`" as binary functor
  - first argument: first element of list ("*head*")
  - second argument: rest of list ("*tail*")

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

Abbreviating notations for lists

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

  Example: `[a | [b | [c | [d | []]]]]`
- element separator: `[` *element1* `,` *element2* `]`
  `= [` *element1* `| [` *element2* `| []]]`

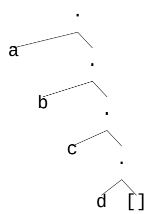
  Example: `[a, b, c, d]`

An example for the four notations

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

           =  [a | [b | [c | [d | []]]]]
```

=



Recursive relations in Prolog
Example relations I: append

- Idea: a relation concatenating two lists
- Example: `?- append([a,b,c],[d,e],X).` ⇒ `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 IIa: (naive) reverse

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

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


Recursive relations in Prolog
Example relations IIb: reverse

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

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


Some practical matters

- To start Prolog on the Linguistics Department Unix machines:
  - SWI-Prolog: `pl`
  - SICStus: `prolog` or `M-x run-prolog` in XEmacs
- At the Prolog prompt (`?-`):
  - Exit Prolog: `halt.`
  - Consult a file in Prolog: `[`*`filename`*`].`[3]
- The manuals are accessible from the course web page.

[3] The `.pl` suffix is added automatically, but use single quotes if name starts with a capital letter or contains special characters such as "." or "–". For example `['MyGrammar'].` or `['~/file-1'].`

Encoding finite state automata in Prolog
What needs to be represented?

A **finite state automaton** is a quintuple (Q, Σ, E, S, F) with

- Q a finite set of states
- Σ a finite set of symbols, the alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- E a set of edges $Q \times (\Sigma \cup \{\epsilon\}) \times Q$

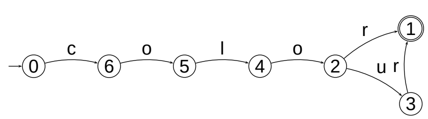
Prolog representation of a finite state automaton

The FSA is represented by the following kind of Prolog facts:

- initial nodes: `initial(`*`nodename`*`).`
- final nodes: `final(`*`nodename`*`).`
- edges: `arc(`*`from-node`*`, `*`label`*`, `*`to-node`*`).`

A simple example

FSTN representation of FSM:

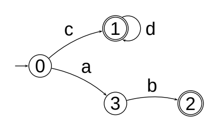


Prolog encoding of FSM:

```
initial(0).
final(1).
arc(0,c,6). arc(6,o,5). arc(5,l,4). arc(4,o,2).
arc(2,r,1). arc(2,u,3). arc(3,r,1).
```


An example with two final states

FSTN representation of FSM:



Prolog encoding of FSM:

```
initial(0).
final(1).    final(2).
arc(0,c,1).  arc(1,d,1).  arc(0,a,3).  arc(3,b,2).
```

Recognition with FSMs in Prolog
fstn_traversal_basic.pl

```
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(First,[First|Rest],Rest).
```


Generation with FSMs in Prolog

```
generate :-
        test(X),
        write(X),
        nl,
        fail.
```


Encoding finite state transducers in Prolog
What needs to be represented?

A **finite state transducer** is a 6-tuple $(Q, \Sigma_1, \Sigma_2, E, S, F)$ with

- Q a finite set of states
- Σ_1 a finite set of symbols, the input alphabet
- Σ_2 a finite set of symbols, the output alphabet
- $S \subseteq Q$ the set of start states
- $F \subseteq Q$ the set of final states
- E a set of edges $Q \times (\Sigma_1 \cup \{\epsilon\}) \times Q \times (\Sigma_2 \cup \{\epsilon\})$

Prolog representation of a transducer

The only change compared to automata, is an additional argument in the representation of the arcs:

`arc(`*`from-node`*`, `*`label-in`*`, `*`to-node`*`, `*`label-out`*`).`

Example:

```
initial(1).
final(5).
arc(1,2,where,ou).
arc(2,3,is,est).
arc(3,4,the,la).
arc(4,5,exit,sortie).
arc(4,5,shop,boutique).
arc(4,5,toilet,toilette).
arc(3,6,the,le).
arc(6,5,policeman,gendarme).
```


Processing with a finite state transducer

```
test(Input,Output) :-
   initial(Node),
   transduce(Node,Input,Output),
   write(Output),nl.

transduce(Node,[],[]) :-
   final(Node).

transduce(Node1,String1,String2) :-
   arc(Node1,Node2,Label1,Label2),
   traverse2(Label1,Label2,String1,NewString1,
                           String2,NewString2),
   transduce(Node2,NewString1,NewString2).

traverse2(Word1,Word2,[Word1|RestString1],RestString1,
                      [Word2|RestString2],RestString2).
```


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) :-
    \+ special(Label).
traverse(Abbrev,[Label|RestString],RestString) :-
    macro(Abbrev,Label).
traverse('#',String,String).

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


A tiny English fragment as an example
(fsa/ex_simple_engl.pl)

```
initial(1).        arc(7,n,9).        macro(n,man).
final(9).          arc(8,adj,9).      macro(n,woman).
arc(1,np,3).       arc(8,mod,8).      macro(pv,is).
arc(1,det,2).      arc(9,cnj,4).      macro(pv,was).
arc(2,n,3).        arc(9,cnj,1).      macro(cnj,and).
arc(3,pv,4).                          macro(cnj,or).
arc(4,adv,5).      macro(np,kim).     macro(adj,happy).
arc(4,'#',5).      macro(np,sandy).   macro(adj,stupid).
arc(5,det,6).      macro(np,lee).     macro(mod,very).
arc(5,det,7).      macro(det,a).      macro(adv,often).
arc(5,'#',8).      macro(det,the).    macro(adv,always).
arc(6,adj,7).      macro(det,her).    macro(adv,sometimes).
arc(6,mod,6).      macro(n,consumer).
```

Reading assignment

- Pages 1–26 of Fernando Pereira and Stuart Shieber (1987): *Prolog and Natural-Language Analysis*. Stanford: CSLI.