Implementing finite state machines

- A first introduction to PROLOG
- Encoding finite state machines in PROLOG
- Recognition and generation with finite state machines in PROLOG
- Search spaces and how to traverse them
  - depth-first
  - breadth-first
- Encoding finite state transducers in PROLOG

The PROLOG programming language (1)

PROgrammation LOGique invented by Alain Colmerauer and colleagues at Marseille in 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[1]
  - *examples:* `twelve, a, q_1`
- **variables** over entities
  - *syntax*: starting with upper-case letter[2]
  - *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)`

[1] starting with lower-case letter, a number or surrounded by single quotes
[2] starting with upper-case letter or underscore: `[A-Z_][a-zA-Z0-9_]*`

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).`
- **nonunit clauses** or **rules**
  - *syntax:* rel_0 `:-` rel_1`, ...,` rel_n`.`
  - *example:*

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

Note:

- Variables only have scope over a single clause; there are no global variables.
- There is no explicit typing of variables or of the arguments of predicates.

Some practical matters

- Add a separate line with the word "COMPLING" (and a return at the end) to your file `~/.subscriptions`.
- Start PROLOG (on the Ling. Dep. UNIX machines)
  - Either at UNIX prompt: `prolog`
  - Or in Emacs: `M-x run-prolog`
- At the PROLOG prompt (`?-`):
  - Exit PROLOG: `exit.`
  - Consult a file in PROLOG: `[`*`filename`*`].`[3]
  - Obtain more solutions: `;`
- The SICStus manual is accessible from the course web page.

[3] The `.pl` ending is added automatically, but you need to use single quotes around the filename if it starts with a capital letter or contains special characters such as "." or "–". For example `['MyGrammar'].` or `['~/file-1'].`

Tracing a prolog query

An example program:

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

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

Tracing a query:

```
 ?- trace.
{The debugger will first creep -- showing everything (trace)
yes
| ?- grandfather(adam,X).
        1      1 Call: grandfather(adam,_218) ?
        2      2 Call: father(adam,_675) ?
        2      2 Exit: father(adam,ben) ?
        3      2 Call: father(ben,_218) ?
?       3      2 Exit: father(ben,claire) ?
?       1      1 Exit: grandfather(adam,claire) ?

X = claire ? ;
```



```
        1      1 Redo: grandfather(adam,claire) ?
        3      2 Redo: father(ben,claire) ?
        3      2 Exit: father(ben,chris) ?
        1      1 Exit: grandfather(adam,chris) ?

X = chris ? ;

no
```


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

We need to represent:

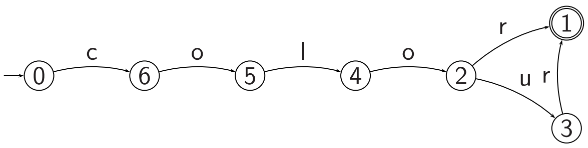
- initial nodes
- final nodes
- edges

Represented as facts in PROLOG:

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

A simple example

FSTN representation of FSM:

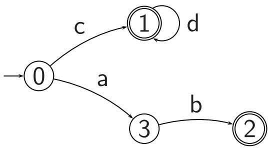


PROLOG encoding of FSM:

```
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

```
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).
```


Tracing the two final state example

```
| ?- test([c,d]).
        1      1 Call: test([c,d]) ?

        2      2 Call: initial(_681) ?
        2      2 Exit: initial(0) ?
        3      2 Call: recognize(0,[c,d]) ?
        4      3 Call: arc(0,_1805,_1806) ?
?       4      3 Exit: arc(0,c,1) ?
        5      3 Call: traverse(c,[c,d],_1799) ?
        5      3 Exit: traverse(c,[c,d],[d]) ?
        6      3 Call: recognize(1,[d]) ?
        7      4 Call: arc(1,_3686,_3687) ?
        7      4 Exit: arc(1,d,1) ?
        8      4 Call: traverse(d,[d],_3680) ?
        8      4 Exit: traverse(d,[d],[]) ?
        9      4 Call: recognize(1,[]) ?
       10      5 Call: final(1) ?
       10      5 Exit: final(1) ?
?       9      4 Exit: recognize(1,[]) ?
?       6      3 Exit: recognize(1,[d]) ?
?       3      2 Exit: recognize(0,[c,d]) ?
?       1      1 Exit: test([c,d]) ?

yes
{trace,source_info}
```

Another example trace

```
| ?- test(X).
        1      1 Call: test(_206) ?
        2      2 Call: initial(_656) ?
        2      2 Exit: initial(0) ?
        3      2 Call: recognize(0,_206) ?
        4      3 Call: final(0) ?
        4      3 Fail: final(0) ?
        5      3 Call: arc(0,_1778,_1779) ?
?       5      3 Exit: arc(0,c,1) ?
        6      3 Call: traverse(c,_206,_1772) ?
        6      3 Exit: traverse(c,[c|_1772],_1772) ?
        7      3 Call: recognize(1,_1772) ?
        8      4 Call: final(1) ?
        8      4 Exit: final(1) ?
?       7      3 Exit: recognize(1,[]) ?
?       3      2 Exit: recognize(0,[c]) ?
?       1      1 Exit: test([c]) ?

X = [c] ? ;
        1      1 Redo: test([c]) ?
        3      2 Redo: recognize(0,[c]) ?
        7      3 Redo: recognize(1,[]) ?
        9      4 Call: arc(1,_3654,_3655) ?
```



```
        9      4 Exit: arc(1,d,1) ?
       10      4 Call: traverse(d,_1772,_3648) ?
       10      4 Exit: traverse(d,[d|_3648],_3648) ?
       11      4 Call: recognize(1,_3648) ?
       12      5 Call: final(1) ?
       12      5 Exit: final(1) ?
?      11      4 Exit: recognize(1,[]) ?
?       7      3 Exit: recognize(1,[d]) ?
?       3      2 Exit: recognize(0,[c,d]) ?
?       1      1 Exit: test([c,d]) ?

X = [c,d] ? ;
        1      1 Redo: test([c,d]) ?
        3      2 Redo: recognize(0,[c,d]) ?
        7      3 Redo: recognize(1,[d]) ?
       11      4 Redo: recognize(1,[]) ?
       13      5 Call: arc(1,_5524,_5525) ?
       13      5 Exit: arc(1,d,1) ?
       14      5 Call: traverse(d,_3648,_5518) ?
       14      5 Exit: traverse(d,[d|_5518],_5518) ?
       15      5 Call: recognize(1,_5518) ?
       16      6 Call: final(1) ?
       16      6 Exit: final(1) ?
?      15      5 Exit: recognize(1,[]) ?
?      11      4 Exit: recognize(1,[d]) ?
?       7      3 Exit: recognize(1,[d,d]) ?
```



```
?       3      2 Exit: recognize(0,[c,d,d]) ?
?       1      1 Exit: test([c,d,d]) ?

X = [c,d,d] ? ;
        1      1 Redo: test([c,d,d]) ?
        3      2 Redo: recognize(0,[c,d,d]) ?
        7      3 Redo: recognize(1,[d,d]) ?
       11      4 Redo: recognize(1,[d]) ?
       15      5 Redo: recognize(1,[]) ?
       17      6 Call: arc(1,_7394,_7395) ?
       17      6 Exit: arc(1,d,1) ?
       18      6 Call: traverse(d,_5518,_7388) ?
       18      6 Exit: traverse(d,[d|_7388],_7388) ?
       19      6 Call: recognize(1,_7388) ?
       20      7 Call: final(1) ?
       20      7 Exit: final(1) ?
?      19      6 Exit: recognize(1,[]) ?
?      15      5 Exit: recognize(1,[d]) ?
?      11      4 Exit: recognize(1,[d,d]) ?
?       7      3 Exit: recognize(1,[d,d,d]) ?
?       3      2 Exit: recognize(0,[c,d,d,d]) ?
?       1      1 Exit: test([c,d,d,d]) ?

X = [c,d,d,d] ?
```


Generation with FSMs in PROLOG

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

Encoding finite state transducers in PROLOG

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

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

transduce(Node_1,String1,String2) :-
   arc(Node_1,Node_2,Label1,Label2),
   traverse2(Label1,Label2,
             String1,NewString1,
             String2,NewString2),
   transduce(Node_2,NewString1,NewString2).

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


An example for a transducer

```
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).
```


An example trace

```
| ?- test([where,is,the,exit],Output).
        1      1 Call: test([where,is,the,exit],_274) ?
        2      2 Call: initial(_775) ?
        2      2 Exit: initial(1) ?
        3      2 Call: transduce(1,[where,is,the,exit],_274) ?
        4      3 Call: arc(1,_1909,_1910,_1911) ?
        4      3 Exit: arc(1,2,where,ou) ?
        5      3 Call: traverse2(where,ou,[where,is,the,exit],_1901,_274,_1903) ?
        5      3 Exit: traverse2(where,ou,[where,is,the,exit],[is,the,exit],[ou|_1903]
        6      3 Call: transduce(2,[is,the,exit],_1903) ?
        7      4 Call: arc(2,_3814,_3815,_3816) ?
        7      4 Exit: arc(2,3,is,est) ?
        8      4 Call: traverse2(is,est,[is,the,exit],_3806,_1903,_3808) ?
        8      4 Exit: traverse2(is,est,[is,the,exit],[the,exit],[est|_3808],_3808) ?
        9      4 Call: transduce(3,[the,exit],_3808) ?
       10      5 Call: arc(3,_5715,_5716,_5717) ?
?      10      5 Exit: arc(3,4,the,la) ?
       11      5 Call: traverse2(the,la,[the,exit],_5707,_3808,_5709) ?
       11      5 Exit: traverse2(the,la,[the,exit],[exit],[la|_5709],_5709) ?
       12      5 Call: transduce(4,[exit],_5709) ?
       13      6 Call: arc(4,_7618,_7619,_7620) ?
?      13      6 Exit: arc(4,5,exit,sortie) ?
       14      6 Call: traverse2(exit,sortie,[exit],_7610,_5709,_7612) ?
       14      6 Exit: traverse2(exit,sortie,[exit],[],[sortie|_7612],_7612) ?
       15      6 Call: transduce(5,[],_7612) ?
       16      7 Call: final(5) ?
       16      7 Exit: final(5) ?
?      15      6 Exit: transduce(5,[],[]) ?
?      12      5 Exit: transduce(4,[exit],[sortie]) ?
?       9      4 Exit: transduce(3,[the,exit],[la,sortie]) ?
?       6      3 Exit: transduce(2,[is,the,exit],[est,la,sortie]) ?
?       3      2 Exit: transduce(1,[where,is,the,exit],[ou,est,la,sortie]) ?
       17      2 Call: write([ou,est,la,sortie]) ?
[ou,est,la,sortie       17      2 Exit: write([ou,est,la,sortie]) ? ]
       18      2 Call: nl ?

       18      2 Exit: nl ?
?       1      1 Exit: test([where,is,the,exit],[ou,est,la,sortie]) ?

Output = [ou,est,la,sortie] ? ;
```



```
        1      1 Redo: test([where,is,the,exit],[ou,est,la,sortie]) ?
        3      2 Redo: transduce(1,[where,is,the,exit],[ou,est,la,sortie]) ?
        6      3 Redo: transduce(2,[is,the,exit],[est,la,sortie]) ?
        9      4 Redo: transduce(3,[the,exit],[la,sortie]) ?
       12      5 Redo: transduce(4,[exit],[sortie]) ?
       15      6 Redo: transduce(5,[],[]) ?
       19      7 Call: arc(5,_9517,_9518,_9519) ?
       19      7 Fail: arc(5,_9517,_9518,_9519) ?
       15      6 Fail: transduce(5,[],_7612) ?
       13      6 Redo: arc(4,5,exit,sortie) ?
?      13      6 Exit: arc(4,5,shop,boutique) ?
       20      6 Call: traverse2(shop,boutique,[exit],_7610,_5709,_7612) ?
       20      6 Fail: traverse2(shop,boutique,[exit],_7610,_5709,_7612) ?
       13      6 Redo: arc(4,5,shop,boutique) ?
       13      6 Exit: arc(4,5,toilet,toilette) ?
       21      6 Call: traverse2(toilet,toilette,[exit],_7610,_5709,_7612) ?
       21      6 Fail: traverse2(toilet,toilette,[exit],_7610,_5709,_7612) ?
       12      5 Fail: transduce(4,[exit],_5709) ?
       10      5 Redo: arc(3,4,the,la) ?
       10      5 Exit: arc(3,6,the,le) ?
       22      5 Call: traverse2(the,le,[the,exit],_5707,_3808,_5709) ?
       22      5 Exit: traverse2(the,le,[the,exit],[exit],[le|_5709],_5709) ?
       23      5 Call: transduce(6,[exit],_5709) ?
       24      6 Call: arc(6,_7612,_7613,_7614) ?
       24      6 Exit: arc(6,5,policeman,gendarme) ?
       25      6 Call: traverse2(policeman,gendarme,[exit],_7604,_5709,_7606) ?
       25      6 Fail: traverse2(policeman,gendarme,[exit],_7604,_5709,_7606) ?
       23      5 Fail: transduce(6,[exit],_5709) ?
        9      4 Fail: transduce(3,[the,exit],_3808) ?
        6      3 Fail: transduce(2,[is,the,exit],_1903) ?
        3      2 Fail: transduce(1,[where,is,the,exit],_274) ?
        1      1 Fail: test([where,is,the,exit],_274) ?

no
{trace,source_info}
| ?-
```

A non-determinstic example trace

```
| ?- test([where,is,the,_],Output).
        1      1 Call: test([where,is,the,_247],_278) ?
        2      2 Call: initial(_779) ?
        2      2 Exit: initial(1) ?
        3      2 Call: transduce(1,[where,is,the,_247],_278) ?
        4      3 Call: arc(1,_1913,_1914,_1915) ?
        4      3 Exit: arc(1,2,where,ou) ?
        5      3 Call: traverse2(where,ou,[where,is,the,_247],_1905,_278,_1907) ?
        5      3 Exit: traverse2(where,ou,[where,is,the,_247],[is,the,_247],[ou|_1907]
        6      3 Call: transduce(2,[is,the,_247],_1907) ?
        7      4 Call: arc(2,_3818,_3819,_3820) ?
        7      4 Exit: arc(2,3,is,est) ?
        8      4 Call: traverse2(is,est,[is,the,_247],_3810,_1907,_3812) ?
        8      4 Exit: traverse2(is,est,[is,the,_247],[the,_247],[est|_3812],_3812) ?
        9      4 Call: transduce(3,[the,_247],_3812) ?
       10      5 Call: arc(3,_5719,_5720,_5721) ?
?      10      5 Exit: arc(3,4,the,la) ?
       11      5 Call: traverse2(the,la,[the,_247],_5711,_3812,_5713) ?
       11      5 Exit: traverse2(the,la,[the,_247],[_247],[la|_5713],_5713) ?
       12      5 Call: transduce(4,[_247],_5713) ?
       13      6 Call: arc(4,_7622,_7623,_7624) ?
?      13      6 Exit: arc(4,5,exit,sortie) ?
       14      6 Call: traverse2(exit,sortie,[_247],_7614,_5713,_7616) ?
       14      6 Exit: traverse2(exit,sortie,[exit],[],[sortie|_7616],_7616) ?
       15      6 Call: transduce(5,[],_7616) ?
       16      7 Call: final(5) ?
       16      7 Exit: final(5) ?
?      15      6 Exit: transduce(5,[],[]) ?
?      12      5 Exit: transduce(4,[exit],[sortie]) ?
?       9      4 Exit: transduce(3,[the,exit],[la,sortie]) ?
?       6      3 Exit: transduce(2,[is,the,exit],[est,la,sortie]) ?
?       3      2 Exit: transduce(1,[where,is,the,exit],[ou,est,la,sortie]) ?
       17      2 Call: write([ou,est,la,sortie]) ?
[ou,est,la,sortie       17      2 Exit: write([ou,est,la,sortie]) ? ]
       18      2 Call: nl ?

       18      2 Exit: nl ?
?       1      1 Exit: test([where,is,the,exit],[ou,est,la,sortie]) ?

Output = [ou,est,la,sortie] ? ;
```



```
        1      1 Redo: test([where,is,the,exit],[ou,est,la,sortie]) ?
        3      2 Redo: transduce(1,[where,is,the,exit],[ou,est,la,sortie]) ?
        6      3 Redo: transduce(2,[is,the,exit],[est,la,sortie]) ?
        9      4 Redo: transduce(3,[the,exit],[la,sortie]) ?
       12      5 Redo: transduce(4,[exit],[sortie]) ?
       15      6 Redo: transduce(5,[],[]) ?
       19      7 Call: arc(5,_9521,_9522,_9523) ?
       19      7 Fail: arc(5,_9521,_9522,_9523) ?
       15      6 Fail: transduce(5,[],_7616) ?
       13      6 Redo: arc(4,5,exit,sortie) ?
?      13      6 Exit: arc(4,5,shop,boutique) ?
       20      6 Call: traverse2(shop,boutique,[_247],_7614,_5713,_7616) ?
       20      6 Exit: traverse2(shop,boutique,[shop],[],[boutique|_7616],_7616) ?
       21      6 Call: transduce(5,[],_7616) ?
       22      7 Call: final(5) ?
       22      7 Exit: final(5) ?
?      21      6 Exit: transduce(5,[],[]) ?
?      12      5 Exit: transduce(4,[shop],[boutique]) ?
?       9      4 Exit: transduce(3,[the,shop],[la,boutique]) ?
?       6      3 Exit: transduce(2,[is,the,shop],[est,la,boutique]) ?
?       3      2 Exit: transduce(1,[where,is,the,shop],[ou,est,la,boutique]) ?
       23      2 Call: write([ou,est,la,boutique]) ?
[ou,est,la,boutique       23      2 Exit: write([ou,est,la,boutique]) ? ]
       24      2 Call: nl ?

       24      2 Exit: nl ?
?       1      1 Exit: test([where,is,the,shop],[ou,est,la,boutique]) ?

Output = [ou,est,la,boutique] ? ;
        1      1 Redo: test([where,is,the,shop],[ou,est,la,boutique]) ?
        3      2 Redo: transduce(1,[where,is,the,shop],[ou,est,la,boutique]) ?
        6      3 Redo: transduce(2,[is,the,shop],[est,la,boutique]) ?
        9      4 Redo: transduce(3,[the,shop],[la,boutique]) ?
       12      5 Redo: transduce(4,[shop],[boutique]) ?
       21      6 Redo: transduce(5,[],[]) ?
       25      7 Call: arc(5,_9521,_9522,_9523) ?
       25      7 Fail: arc(5,_9521,_9522,_9523) ?
       21      6 Fail: transduce(5,[],_7616) ?
```



```
       13      6 Redo: arc(4,5,shop,boutique) ?
       13      6 Exit: arc(4,5,toilet,toilette) ?
       26      6 Call: traverse2(toilet,toilette,[_247],_7614,_5713,_7616) ?
       26      6 Exit: traverse2(toilet,toilette,[toilet],[],[toilette|_7616],_7616) ?
       27      6 Call: transduce(5,[],_7616) ?
       28      7 Call: final(5) ?
       28      7 Exit: final(5) ?
?      27      6 Exit: transduce(5,[],[]) ?
?      12      5 Exit: transduce(4,[toilet],[toilette]) ?
?       9      4 Exit: transduce(3,[the,toilet],[la,toilette]) ?
?       6      3 Exit: transduce(2,[is,the,toilet],[est,la,toilette]) ?
?       3      2 Exit: transduce(1,[where,is,the,toilet],[ou,est,la,toilette]) ?
       29      2 Call: write([ou,est,la,toilette]) ?
[ou,est,la,toilette       29      2 Exit: write([ou,est,la,toilette]) ? ]
       30      2 Call: nl ?

       30      2 Exit: nl ?
?       1      1 Exit: test([where,is,the,toilet],[ou,est,la,toilette]) ?

Output = [ou,est,la,toilette] ? ;

...
```


Background reading assignment

Pages 9–26 (available in shelf in room 201) of:

- F.C.M. Pereira and S.M. Shieber (1987): *Prolog and Natural-Language Analysis*. Stanford: CSLI.