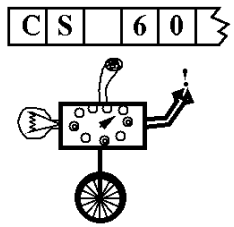



# Finite State Machines

November 26, 2001

## Review: Functions vs. Relations

- Two ways to view list appending:
  - A two-argument function that, given two lists as input, returns a longer list as its result.
  - A *three-place* relation that, given three lists as input, returns true when the third list is a concatenation of the first two.
- Relations are more general than functions.
  - Sometimes mathematicians *define* functions as relations satisfying certain properties.

## Review: Relations vs. Functions

- Ordinary functions are implemented in Prolog as relations.

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

- Although we can think of the first two lists as the input and the last list as an output, e.g.,

```
append([1,2],[3,4,5],X)
```

this is not required! We could just as well ask

```
append([1,2],X,[1,2,3,4,5])
```

or

```
append(X,Y,[1,2,3,4,5])
```

Constraint Solving with Generate & Test:

## "Zebra" Problem

There are five consecutive houses, each of a different color and inhabited by men of different nationalities. They each own a different pet, have a different favorite drink and drive a different car.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right of the ivory house.
6. The Porsche driver owns snails.
7. The Masserati is driven by the man who lives in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who drives a Saab lives in the house next to the man with the fox.
11. The Masserati is driven by the man in the house next to the house where the horse is kept.
12. The Honda driver drinks orange juice.
13. The Japanese drives a Jaguar.
14. The Norwegian lives next to the blue house.

The problem is: Who owns the Zebra? Who drinks water?

## Prolog Solution to “Zebra” (1)

```
left_right(L,R,[L,R,_,_,_]).
left_right(L,R,[_,L,R,_,_]).
left_right(L,R,[_,_,L,R,_]).
left_right(L,R,[_,_,_,L,R]).

next_to(X,Y,L) :-
      left_right(X,Y,L).
next_to(X,Y,L) :-
      left_right(Y,X,L).
```

## Prolog Solution to “Zebra” (2)

```
zebra(S) :-
  S = [[_,norwegian,_,_,_],_,[_,_,_,milk,_],_,_],
   next_to([_,norwegian,_,_,_],[blue,_,_,_,_],S),
   member([green,_,_,coffee,_],S),
   left_right([ivory,_,_,_,_],[green,_,_,_,_],S),
   member([red,englishman,_,_,_],S),
   member([_,ukranian,_,tea,_],S),
   member([yellow,_,masserati,_,_],S),
   member([_,_,honda,orange_juice,_],S),
   member([_,japanese,jaguar,_,_],S),
   member([_,spaniard,_,_,dog],S),
   next_to([_,_,masserati,_,_],[_,_,_,_,horse],S),
   member([_,_,porsche,_,snails],S),
   next_to([_,_,saab,_,_],[_,_,_,_,fox],S).
```

## Warning: Builtin Arithmetic is Not Reversible

- Consider a clause
  `p(X, Y) :- Y is X+1.`
- A possible use of this clause is:
  ```
  | ?-  p(5, Z).
  Z = 6
  ```
- This clause cannot be used in reverse because the is predicate is not reversible:
  ```
  | ?- p(X, 6).
  ```
  will not give `X = 5`; it will fail.

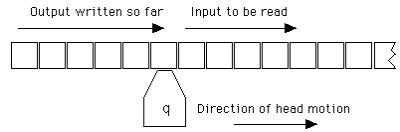
## Review: States and Transitions

- Recall:
  - The *state* of a computation is everything we need to remember in order to know what to do next.
  - We can describe computations as a sequence of states, with *transitions* from each state to the next
  - Problems can be described by giving all possible states and the transitions between them.

# Finite-State Machines

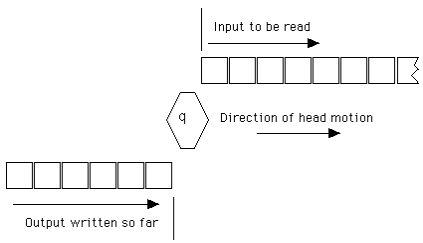
- If the set of all possible states (and transitions) is finite, then we've specified a *finite-state machine*.
  - Mathematical, rather than mechanical (like Turing machines)
- A primitive computational model, related to many facets of computing:
  - Logic circuits having (finite) *memory*
  - The building blocks for most real-life computers
  - Parsers for a limited family of languages (called “regular” languages or finite-state languages)
  - Regular expressions: used for textual pattern matching
  - Real-time software applications
  - The "program" of a Turing machine

# FSM as a “crippled” TM

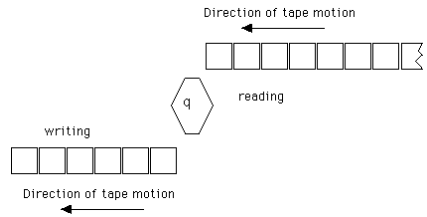



Can move in one direction only, symbol written is never again changed.

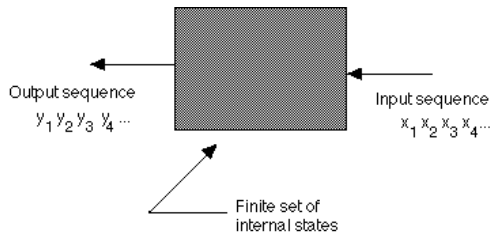
# FSM with separate I/O



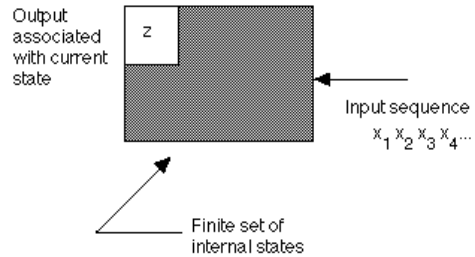

# FSM as head-stationary, tape-moving, device



## FSM as sequence transducer




## FSM as Classifier

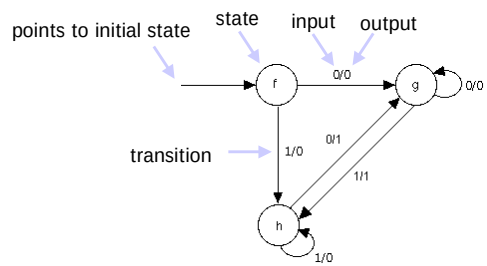



## Edge-Detector Example

| input | output |
|---|---|
| 0 | 0 |
| 00 | 00 |
| 01 | 01 |
| 011 | 010 |
| 0111 | 0100 |
| 01110 | 01001 |

"edges" detections

## Transducer: An Edge Detector



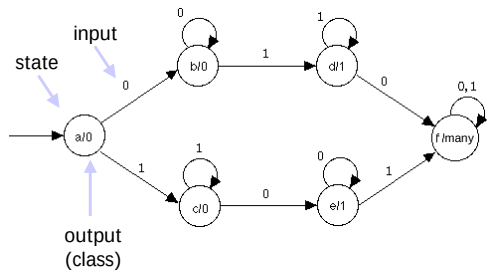

The state is recording the previous input.

Whenever the current input differs, an edge has been detected.

Here, the first input is never considered an edge.

# A related Classifier:

How many edges were there so far (0, 1, many)

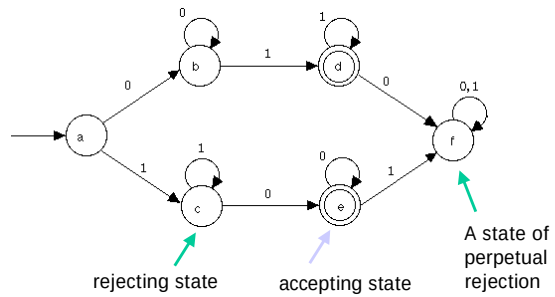



# Acceptors

- Acceptors are Classifiers with only 2 classes: accepted and rejected
  - Rejection is not necessarily final; additional input might move machine to an excepting state
- Typically acceptors drawn with accepting states being a double circle and rejecting states being a single circle

# A related Acceptor:

Accept sequences with exactly one edge



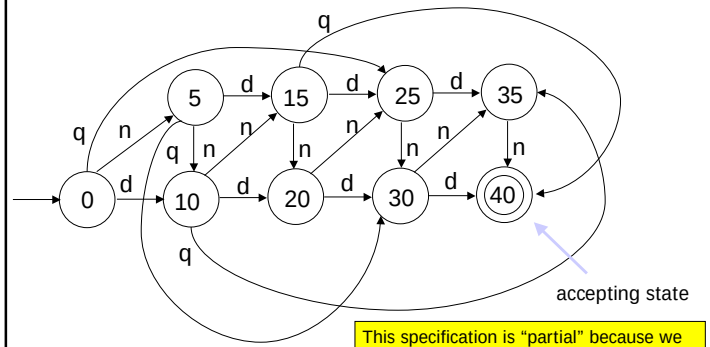

# Conversions

- Every classifier can be represented as a “gang” of acceptors
  - e.g., an acceptor for sequences with no edges, an acceptor for sequences with exactly one edge, and an acceptor for sequences with "many" edges
- Every transducer can be represented as an “equivalent” classifier
- Therefore, studying acceptors, the simplest model, yields insight for all finite-state machines

## What can an Acceptor Do?

- Consider the Pepsi Machine near B101.
- Coins of 5, 10, and 25 cents can be entered
  - Referred to by input symbols n, d, q, respectively.
- Accepts when a total of 40 cents (or more) has been entered.

## Pepsi Acceptor (partial)




This specification is "partial" because we have not said what happens when q is input to states 20, 25, 30, 35, etc.

## Next Lecture

- When discussing grammars we defined a language to be a set of strings.
- The set of inputs (finite sequences) leading to accepting states is a language.
  - Languages accepted by FSMs are called regular languages, and turn out to be very useful.
  - Not all useful languages are regular, however.