finite state automata

- a finite state automaton \((\Sigma, S, s_0, \delta, F)\) is a representation of a machine as a
  - finite set of states \(S\)
  - a state transition relation/table \(\delta\)
    - mapping current state & input symbol from alphabet \(\Sigma\) to the next state
  - an initial state \(s_0\)
  - a set of final states \(F\)
accepting an input

- a fsa accepts an input sequence from an alphabet $\Sigma$ if, starting in the designated starting state, scanning the input sequence leaves the automaton in a final state
- sometimes called recognition
- e.g. automaton that accepts strings of x’s and y’s with an even number of x’s and an odd number of y’s

example

- automaton that accepts strings of x’s and y’s with an even number of x’s and an odd number of y’s
- idea: keep track of whether we have seen even number of x’s and y’s
- $S = \{ee, eo, oe, oo\}$
- $s_0 = ee$
- $\delta = \{(ee, x, oe), (ee, y, eo), \ldots\}$
- $F = \{eo\}$
implementation

- fsa(Input) succeeds if and only if the fsa accepts or recognizes the sequence (list) Input.
- initial state represented by a predicate
  - initial_state(State)
- final states represented by a predicate
  - final_states(List)
- state transition table represented by a predicate
  - next_state(State, InputSymbol, NextState)
- note: next_state need not be a function

implementing fsa/1

- fsa(Input) :- initial_state(S), scan(Input, S).
  % scan is a Boolean predicate

- scan([], State) :- final_states(F), member(State, F).
- scan([Symbol | Seq], State) :- next_state (State, Symbol, Next), scan(Seq, Next).
result propagation

- scan uses pumping/result propagation
- carries around current state and remainder of input sequence
- if FSA is deterministic, when end of input is reached, can make an accept/reject decision immediately; tail recursion optimization can be applied
- if FSA is nondeterministic, may have to backtrack; must keep track of remaining alternatives on execution stack

non-determinism

- a non-deterministic fsa accepts an input sequence if there exists at least one sequence which leaves the automaton in one of its final states
- \(\text{- }\) fsa(Input).
- scan searches through all possible choices for Symbol at each state;
- fails only if no sequence leads to a final state
representing tables

- can use binary connector, e.g., A-B-C instead of next_state(A,B,C)
  - reduces typing;
  - can make it easier to check for errors
- ee-x-oe. ee-y-eo.
- oe-x-ee. oe-y-oo.
- etc.

revised version

scan([], State) :- final_states(F),
    member(State, F).
scan([Symbol | Seq], State) :-
    State-Symbol-Next,
    scan(Seq, Next).
**divide and conquer**

- algorithm design technique
- key idea: reduce problem to two sub-problems of about equal size
- e.g. mergesort
- tournament example
  minimize number of matches required to fairly determine
  - winner
  - runner-up

**tournament definitions**

- *runner-up* is the winner of a sub-tournament among losers to *winner*

  by definition, *winner* has not lost any tournament match

  losers to *winner* are all themselves winners except for the loser of the winner's 1st game

  so we don't need a sub-tournament among all other players, just those who lost to *winner*
minimum matches

- minimum matches required to determine winner = n - 1
- why?
  - every one except the winner is eliminated by a loss to someone
  - every loss requires a match
  - n-1 losers implies n-1 matches
- minimum # of matches for the runner-up?

winner's matches

- we only need matches between those who lost to winner
- how many?
- winner need play no more than $\text{ceiling}(\log_2 n)$ matches
  
  proof based on idea that number of matches = length of path from root to leaf of a binary tree containing n nodes
  shortest path is in a balanced tree
total # of matches

- total matches =
  - matches to determine winner = \( n - 1 \)
  - matches to determine runner-up =
    \( n - 1 + \log_2 n - 1 \)
    \( n + \log_2 n - 2 \)

implementing a round

\[
\text{round}([X], X).
\text{round}([C1, C2], \text{Winner}) :-
    \text{match}(C1, C2, \text{Winner}).
\text{round}(\text{Field}, \text{Winner}) :-
    \text{split}(\text{Field}, \text{Group1}, \text{Group2}),
    \text{round}(\text{Group1}, \text{Winner1}),
    \text{round}(\text{Group2}, \text{Winner2}),
    \text{match}(	ext{Winner1}, \text{Winner2}, \text{Winner}).
\]

- are rules ordered as expected?
  yes -- from specific to general
fixing the match

- can use binary connector
  Competitor-LoserList

match(C1-L1, C2-_, C1-[C2-[] | L1]) :- order(C1, C2).
match(C1-_, C2-L2, C2-[C1-[] | L2]) :- not order(C1, C2).

defining a tournament

tournament(Field, Winner, RunnerUp) :-
  round(Field, Winner-Runners),
  round(Runners, RunnerUp-_).
parsing text and definite clause grammars

Prolog representation for parsing text

- want to parse natural language text
- one way to represent grammar rules:
  sentence --> noun_phrase, verb_phrase.
  stands for
  sentence(X):- append(Y,Z,X),
  noun_phrase(Y), verb_phrase(Z).
- determiner --> [the].
  stands for
  determiner([the]).
- must guess how to split the sequence, inefficient; let constituent parsers decide
a better representation

- sentence(S0,S):-
  noun_phrase(S0,S1), verb_phrase(S1,S).
- determiner([the | S],S).
- 1st argument is sequence to parse and 2nd argument is what is left after removing it
- Rule means “there is a sentence between S0 and S if ...”
- ?-sentence([the, boy, drinks, the, juice], []). succeeds
- ?-noun_phrase([the, boy, drinks, the, juice], R). succeeds with R = [drinks, the, juice]

definite clause grammar (DCG) notation

sentence --> noun_phrase, verb_phrase.
stands for
sentence(S0,S):- noun_phrase(S0,S1), verb_phrase(S1,S).
determiner --> [the].
stands for
determiner([the|S],S).
enforcing constraints between constituents

- suppose we want to enforce number agreement
- can add extra argument to pass this info between constituents
- noun_phrase(N) --> determiner(N), noun(N).
- noun(singular) --> [boy].
- noun(plural) --> [boys].
- determiner(singular) --> [a].
- ?- noun_phrase(N,[a, boys],[[]]). fails
- ?- noun_phrase(N,[a, boy],[[]]). succeeds with N = singular

returning a parse tree or interpretation

- Extra arguments can also be used to return a parse tree or interpretation
- noun_phrase(np(D,N)) --> determiner(D), noun(N).
- determiner(determiner(a)) --> [a].
- noun(noun(boy)) --> [boy].
- ?- noun_phrase(PT,[a, boy],[[]]). succeeds with PT = np(determiner(a),noun(boy))
adding extra tests

- can invoke predicates for tests or interpretation by putting between {}
- don’t match input tokens
- e.g. accessing a lexicon
- noun(N,noun(W)) --> [W],
  {is_noun (W,N)}.
- is_noun(boy,singular).

grammar writing tips

- good grammars:
  - are very modular
  - achieve broad coverage with small number of rules
- collecting a corpus of examples can help design and test grammar
- identify patterns built out of certain types of constituents
<table>
<thead>
<tr>
<th>Prolog &amp; text processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>♦ Prolog good for analyzing and generating text</td>
</tr>
<tr>
<td>♦ parsing involves <em>pattern-matching</em></td>
</tr>
<tr>
<td>♦ text &amp; parse-trees are <em>recursive</em> data structures</td>
</tr>
<tr>
<td>♦ text patterns involve <em>many alternatives</em>, backtracking is helpful</td>
</tr>
<tr>
<td>♦ <em>steadfast</em> predicates can analyze and generate</td>
</tr>
</tbody>
</table>

modeling and analyzing concurrent processes
**process algebra**

- Concurrent programs are hard to implement correctly
- Many subtle non-local interactions
- Deadlock occurs when some processes are blocked forever waiting for each other
- Process algebra are used to model and analyze concurrent processes

**deadlocking system example**

```plaintext
defproc(deadlockingSystem, user1 | user2 $ lock1s0 | lock2s0 | iterDoSomething).

defproc(user1, acquireLock1 > acquireLock2 > doSomething > releaseLock2 > releaseLock1).

defproc(user2, acquireLock2 > acquireLock1 > doSomething > releaseLock1 > releaseLock2).
```
deadlocking system example

```lisp
defproc(lock1s0, 
   acquireLock1 > lock1s1 ? 0).

defproc(lock1s1, releaseLock1 > lock1s0).

defproc(lock2s0, 
   acquireLock2 > lock2s1 ? 0).

defproc(lock2s1, releaseLock2 > lock2s0).

defproc(iterDoSomething, 
   doSomething > iterDoSomething ? 0).
```

transition relation

- **P – A – RP** means that P can do a *single step* by doing action A and leaving program RP remaining
- **empty program**: 0 – A – P is always false.
- **primitive action**: A – A – 0 holds, i.e., an action that has completed leaves nothing more to be done.
- **sequence**: (A > P) – A – P
- **nondeterministic choice**: (P₁ ? P₂) – A – P holds if either P₁ – A – P holds or P₂ – A – P holds.
transition relation

- **interleaved concurrency**: \((P_1 \mid P_2) - A - P\) holds if either \(P_1 - A - P_{11}\) holds and \(P = (P_{11} \mid P_2)\), or \(P_2 - A - P_{21}\) holds and \(P = (P_1 \mid P_{21})\).

- **synchronized concurrency**: \((P_1 \& P_2) - A - P\) holds if both \(P_1 - A - P_{11}\) holds and \(P_2 - A - P_{21}\) holds and \(P = (P_{11} \& P_{21})\).

- **recursive procedures**: \(\text{ProcName} - A - P\) holds if \(\text{ProcName}\) is the name of a procedure that has body \(B\) and \(B - A - P\) holds.

can check properties by searching process graph

- a process has an *infinite execution* if there is a cycle in its configuration graph
- e.g. `defproc(aloop, a > aloop)`
- `has_infinite_run(P):- P - _ - PN, has_infinite_run(PN,[P]).`
- `has_infinite_run(P,V):- member(P,V), !.`
- `has_infinite_run(P,V):- P - _ - PN, has_infinite_run(PN,[P|V]).`
checking properties by searching process graph

- cannot_occur(P,A) holds if no execution of P where action A occurs
- search graph for a transition P1 - A - P2
- useful built-in predicate: forall(+Cond, +Action) holds iff for all bindings of Cond, Action succeeds
- e.g. forall(member(C,[8,3,9]), C >= 3) succeeds

cannot_occur examples

- ?- cannot_occur(a > b | a > c, b). succeeds or fails?
- ?- cannot_occur((a > b | a > c)$ (a > c), b). succeeds or fails?
whenever_eventually

- whenever_eventually(P,A1,A2) holds if in all executions of P whenever action A1 occurs, action A occurs afterwards.
- ?- whenever_eventually(a > b > a, a, b). succeeds or fails?
- ?- whenever_eventually(a > b | a > c, a, b). succeeds or fails?

whenever_eventually examples

- ?- whenever_eventually(loop1, a, b). succeeds or fails, where defproc(loop1, a > b > loop1)?
- ?- whenever_eventually(loop1, b, a). succeeds or fails, where defproc(loop1, a > b > loop1)?
- ?- whenever_eventually(loop2, b, a). succeeds or fails, where defproc(loop2, a > b > (loop2 ? 0)).
**deadlock_free**

- deadlock_free(P) holds if process P cannot reach a deadlocked configuration, i.e. one where the remaining process is not final, but no transition is possible.
- ?- deadlock_free(a $ a). succeeds or fails?
- ?- deadlock_free(a > a $ a). succeeds or fails?

**deadlock_free examples**

- ?- deadlock_free(loop3 $ a). where defproc(loop3, (a > loop3) ? 0)) succeeds or fails?