



Computational Linguistics (INF2820 — FSAs)

{ baa!, baaa!, baaaa!, ... }

Stephan Oepen

Universitetet i Oslo

oe@ifi.uio.no

Background: A Bit of Formal Language Theory

Languages as Sets of Utterances

- What is a language? And how can one characterize it (precisely)?
 - simplifying assumption: language as a *set of strings* ('utterances');
- well-formed utterances are set members, ill-formed ones are not;
- provides no account of utterance-internal structure, e.g. 'subject';
- + mathematically very simple, hence computationally straightforward.



Background: A Bit of Formal Language Theory

Languages as Sets of Utterances

- What is a language? And how can one characterize it (precisely)?
 - simplifying assumption: language as a *set of strings* ('utterances');
- well-formed utterances are set members, ill-formed ones are not;
- provides no account of utterance-internal structure, e.g. 'subject';
- + mathematically very simple, hence computationally straightforward.

Regular Expressions

- Even simple languages (e.g. arithmetic expressions) can be infinite;
- to obtain a *finite description* of an infinite set → *regular expressions*.



Brushing Up our Knowledge of Regular Expressions

`/[wW]oodchucks?/`

woodchuck — *Woodchuck* — *woodgrubs* — *woodchucks* — *wood*



Brushing Up our Knowledge of Regular Expressions

`/[wW]oodchucks?/`

woodchuck — *Woodchuck* — *woodgrubs* — *woodchucks* — *wood*

`/baa+!/`

ba! — *baa!* — *baah!* — *baaaa!* — *baaaaaaaaaa!*



Brushing Up our Knowledge of Regular Expressions

`/[wW]oodchucks?/`

woodchuck — *Woodchuck* — *woodgrubs* — *woodchucks* — *wood*

`/baa+!/`

ba! — *baa!* — *baah!* — *baaaa!* — *baaaaaaaaaa!*

aa — *aaa* — *aaaa* — *aaaaaa* — *aaaaaaaa* — *aaaaaaaaa* — ...



Pattern Matching: Finite-State Automata

`/baa+!/`

ba! — baa! — baah! — baaaa! — baaaaaaaaa!



Pattern Matching: Finite-State Automata

/baa+!/

ba! — *baa!* — *baah!* — *baaaa!* — *baaaaaaaaaa!*

Recognizing Regular Languages

- Finite-State Automata (FSAs) are *very restricted* Turing machines;
 - states and transitions: read one symbol at a time from input tape;
- *accept* utterance when no more input, in a ‘final’ state; else *reject*.



A Rather More Complex Example

$/(aa)^+|(aaa)^+/$

aa — aaa — aaaa — aaaaaa — aaaaaaaa — aaaaaaaaa — ...



A Rather More Complex Example

$/(aa)^+|(aaa)^+/$

$aa — aaa — aaaa — aaaaaa — aaaaaaaa — aaaaaaaaa — \dots$

- Non-Deterministic FSAs (NFSAs): multiple transitions per symbol;
→ a *search space* of possible solutions: decisions no longer obvious.



Quite Abstractly: Three Approaches to Search

(Heuristic) Look-Ahead

- Peek at input tape one or more positions beyond the current symbol;
- try to work out (or 'guess') which branch to take for current symbol.

Parallel Computation

- Assume unlimited computational resources, i.e. any number of cpus;
- copy FSA, remaining input, and current state → multiple branches.

Backtracking (Or Back-Up)

- Keep track of possibilities (*choice points*) and remaining candidates;
- 'leave a bread crumb', go down one branch; eventually come back.



NFSA Recognition (From Jurafsky & Martin, 2008)

```
1  procedure nd-recognize(tape , fsa) ≡
2    agenda ← {⟨0, 0⟩};
3    do
4      current ← pop(agenda);
5      state ← first(current);
6      index ← second(current);
7      if (index = length(tape) and state is final state) then
8        return accept;
9      fi
10     for(next in fsa.transitions[state, tape[index]]) do
11       agenda ← agenda ∪ {⟨next, index + 1⟩}
12     od
13     if agenda is empty then return reject; fi
14   od
15 end
```



NFSA Recognition (From Jurafsky & Martin, 2008)

```
1 procedure nd-recognize(tape , fsa) ≡
2   agenda ← {⟨0, 0⟩};
3   do
4     current ← pop(agenda);
5     state ← first(current);
6     index ← second(current);
7     if (index = length(tape) and state is final state) then
8       return accept;
9     fi
10    for(next in fsa.transitions[state, tape[index]]) do
11      agenda ← agenda ∪ {⟨next, index + 1⟩}
12    od
13    if agenda is empty then return reject; fi
14  od
15 end
```

