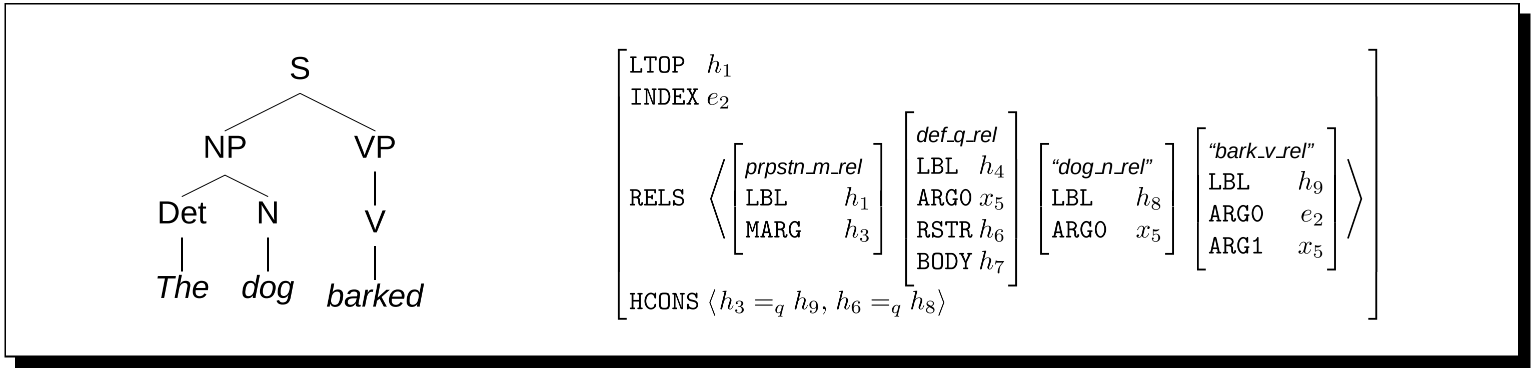

Algorithms for AI and NLP (INF4820 — FSAs)

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

Erik Velldal and Stephan Oepen

Universitetet i Oslo

$\{$`erive`$|$`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! — baaaaaaaaa!

Brushing Up our Knowledge of Regular Expressions

`/[wW]oodchucks?/`

woodchuck — Woodchuck — woodgrubs — woodchucks — wood

`/baa+!/`

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

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!* — *baaaaaaaaa!*

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*.

Tracing the Recognition of a Simple Input

`/baa+!/`

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

Input Tape

0	1	2	3	4
b	*a*	*a*	*a*	*!*

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* — ...

- 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)

```
procedure nd-recognize(tape , fsa) $\equiv$
  agenda $\leftarrow \{\langle 0, 0 \rangle\}$;
  do
    current $\leftarrow$ pop(agenda);
    state $\leftarrow$ first(current);
    index $\leftarrow$ second(current);
    if (index $=$ length(tape) and state *is final state*) then
      return accept;
    fi
    for(next in fsa.transitions[state, tape[index]]) do
      agenda $\leftarrow$ agenda $\cup \{\langle$next, index $+ 1\rangle\}$
    od
    if agenda *is empty* then return reject; fi
  od
end
```