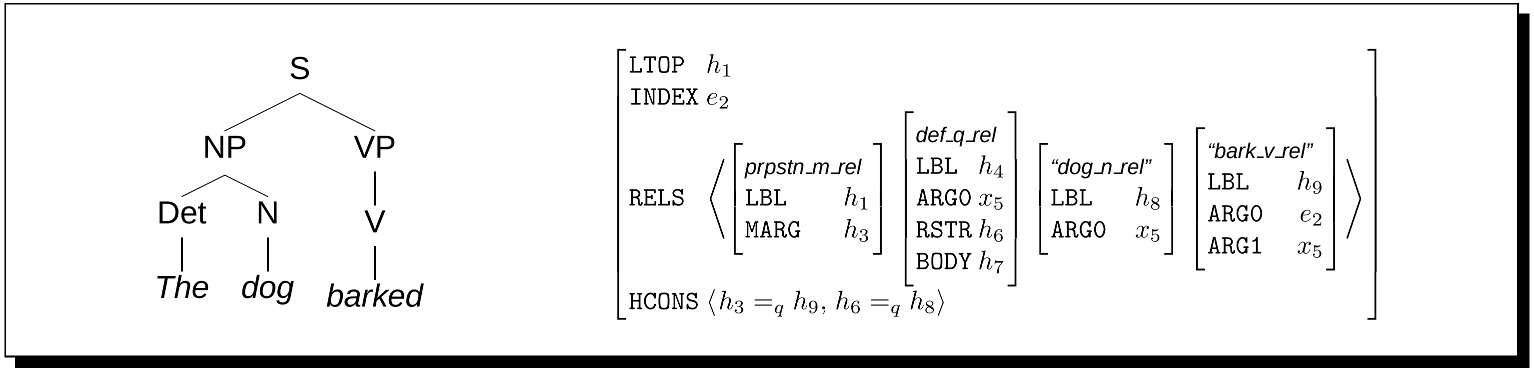

Computational Linguistics (INF2820 — FSAs)

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

Stephan Oepen

Universitetet i Oslo

oe@ifi.uio.no

Defining New Functions

- `defun()` associates a function definition (‘*body*’) with a symbol:

  `(defun` *name* `(`*parameter*$_1$ ... *parameter*$_n$`)` *body*`)`

```
? (defun ! (n)
    (if (equal n 0)
      1
      (* n (! (- n 1)))))
  → !
? (! 0) → 1
? (! 5) → 120
```

- when a function is called, actual arguments (e.g. ‘`0`’ and ‘`5`’) are bound to the function parameter(s) (i.e. ‘`n`’) for the scope of the function body;
- functions evaluate to the value of the *last* sexp in the function *body*.

Recursion as a Control Structure

- A function is said to be *recursive* when its *body* contains a call to itself:

```
(defun mlength (list)
  (if (null list)
    0
    (+ 1 (mlength (rest list)))))
```

- ? `(mlength '(a b)))`

  0: `(MLENGTH (A B))`
    1: `(MLENGTH (B))`
      2: `(MLENGTH NIL)`
      2: returned `0`
    1: returned `1`
  0: returned `2`
  $\longrightarrow$ `2`

- *body* contains (at least) one recursive and one non-recursive branch.

Local Variables

- Sometimes intermediate results need to be accessed more than once;
- `let()` and `let*()` create temporary value bindings for symbols, e.g;

? `(defparameter *foo* 42)` → `*FOO*`

? `(let ((bar (+ *foo* 1))) (* bar 2))` → `86`

? `bar` → *error*

(let ((*variable*$_1$ *sexp*$_1$)
⋮
(*variable*$_n$ *sexp*$_n$))
sexp ... *sexp*)

- bindings valid only in the body of `let()` (other bindings are *shadowed*);
- `let*()` binds *sequentially*, i.e. *variable*$_i$ will be accesible for *variable*$_{i+1}$.

Iteration — Another Control Structure

- Recursion is very powerful, but at times *iteration* comes more natural:

```
(loop
    for number in '(1 2 3 4 5 6 7 8 9)
    when (oddp number)
    collect number))
```

A Selection of `loop()` Directives

- `for` *symbol* { `in` | `on` } *list* iterate *symbol* through *list* elements or tails;
- `for` *symbol* `from` *start* [`to` *end*] [`by` *step*] count *symbol* in range;
- [{ `when` | `unless` } *test*] { `collect` | `append` } *sexp* accumulate *sexp*;
- [`while` *test*] `do` *sexp*$^+$ execute expression(s) *sexp*$^+$ in each iteration.

A Few More Examples

- `loop()` is extremely general; a single iteration construct fits all needs:

  ? `(loop for foo in '(1 2 3) collect foo)`
  → `(1 2 3)`

  ? `(loop for foo on '(1 2 3) collect foo)`
  → `((1 2 3) (2 3) (3))`

  ? `(loop for foo on '(1 2 3) append foo)`
  → `(1 2 3 2 3 3)`

  ? `(loop for i from 1 to 3 by 1 collect i)`
  → `(1 2 3)`

- `loop()` returns the final value of the accumulator (`collect` or `append`);
- `return()` terminates the iteration immediately and returns a value:

  ? `(loop for foo in '(1 2 3) when (evenp foo) do (return foo))`
  → 2

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’);

$\rightarrow$ 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');

$\rightarrow$ 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 $\rightarrow$ *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 on Strings: Finite-State Automata

`/baa+!/`

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

Pattern Matching on Strings: 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;

$\longrightarrow$ *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 $\rightarrow$ 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) ≡
  agenda ← {⟨0, 0⟩};
  do
    current ← pop(agenda);
    state ← first(current);
    index ← 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 ← agenda ∪ {⟨next, index + 1⟩}
    od
    if agenda is empty then return reject; fi
  od
end
```