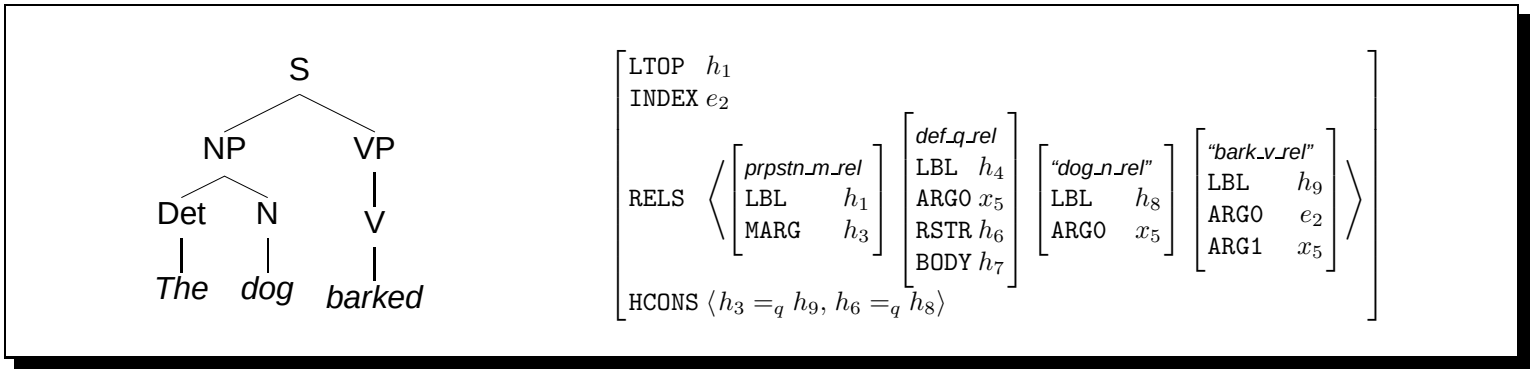


# Algorithms for AI and NLP (INF4820 — More Lisp)

(funcall #'equal 'eq 'equal) → nil

**Stephan Oepen and Jan Tore Lønning**

Universitetet i Oslo

{ oe | jtl }@ifi.uio.no

# HMM Probabilities: The Forward Algorithm

```
1   procedure forward(hmm , input) ≡
2     forward ← new array[N , T]
3     for(1 ≤ c < N − 1) do
4       forward[c , 0] ← P(c|0) ∗ P(input[0]|c)
5     for(1 ≤ t < T) do
6       for(1 ≤ c < N − 1) do
7         for(1 ≤ p < N − 1) do
8           forward[c , t] ← forward[c , t] + forward[p , t − 1] ∗ P(c|p) ∗ P(input[t]|c)
9     for(1 ≤ p < N − 1) do
10      forward[N − 1 , T − 1] ← forward[N − 1 , T − 1] + forward[p , T − 1] ∗ P(N − 1|p)
11    return forward
12  end
```

# Estimating Transition and Emission Probabilities

# Some Objects are More Equal Than Others

- Plethora of equality tests: `eq()`, `eql()`, `equal()`, `equalp()`, and more;
- `eq()` tests object identity; it is not useful for numbers or characters;
- `eql()` is much like `eq()`, but well-defined on numbers and characters;
- `equal()` tests structural equivalence (recursively for lists and strings);

? `(eq (cons 1 (cons 2 (cons 3 nil))) '(1 2 3))` → `nil`

? `(equal (cons 1 (cons 2 (cons 3 nil))) '(1 2 3))` → `t`

? `(eq 42 42)` → *dependent on implementation*

? `(eql 42 42)` → `t`

? `(eql 42 42.0)` → `nil`

? `(equalp 42 42.0)` → `t`

? `(equal "foo" "foo")` → `t`

? `(equalp "FOO" "foo")` → `t`

# Functions as First-Class Citizens

- Built-in functions (`member()`, `position()`, et al.) use `eql()` by default;
- but almost all take an optional `:test` argument to provide a predicate:

  ? `(position "foo" '("foo" "bar"))` → `nil`

  ? `(position "foo" '("foo" "bar") :test #'equal)` → `0`

- `#'foo` is a short-hand for `(function foo)`, yielding a *function object*;
- function objects are first-class citizens, i.e. can be treated just like data;
- `funcall()` can be used to *invoke* a function object on a list of arguments:

  ? `(funcall #'+ 1 2 3)` → `6`

- many built-in functions also take a `:key` argument, an accessor function:

  ? `(sort '((47 :bar) (11 :foo)) #'< :key #'first)`
  → `((11 :FOO) (47 :BAR))`

# Hash Tables: Yet Another Container Data Structure

- Lists are inefficient for indexing data, arrays restricted to numeric keys;
- *hash table* is a (one-dimensional) associative array; free-choice keys;
- any of the four (built-in) equality tests can be used for key comparison;

```
? (defparameter *map* (make-hash-table :test #'equal))
⟶ *map*
? (gethash "foo" *map*) ⟶ nil
? (setf (gethash "foo" *map*) 4711) ⟶ 4711
? (gethash "foo" *map*) ⟶ 4711
```

- the more restricted the test, the more efficient will be hash table access;
- to iterate over hash table entries, use specialize `loop()` directives:

```
? (loop
      for key being each hash-key in *map*
      collect (gethash key *map*))
```

# General Dynamic Programming: Memoization

```
(let ((map (make-hash-table :test #'eql)))
  (defun fibonacci (n)
    (or (gethash n map)
        (setf (gethash n map)
          (cond
           ((= n 0)
            0)
           ((= n 1)
            1)
           (t
            (+ (fibonacci (- n 2)) (fibonacci (- n 1)))))))))
```