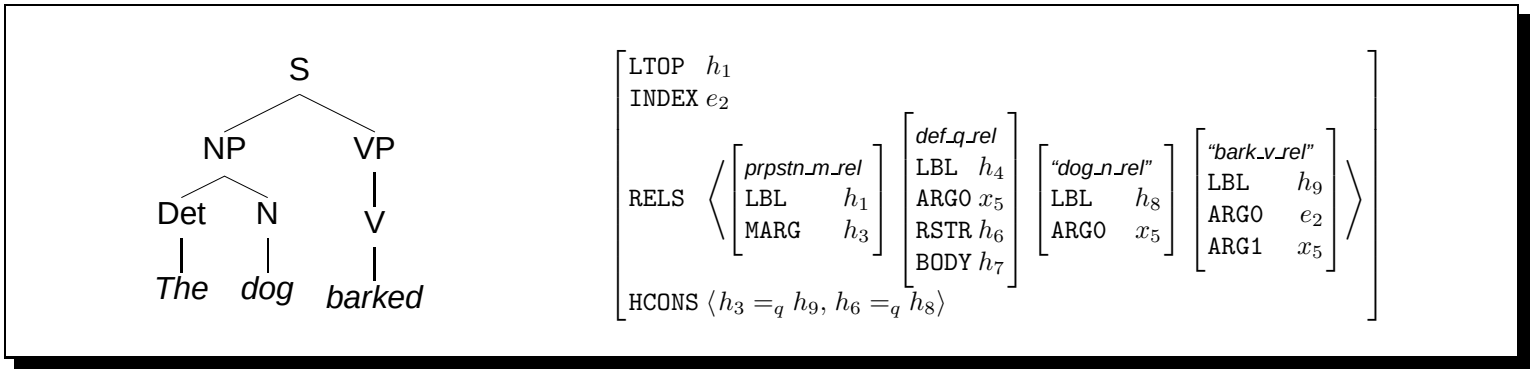

Computational Linguistics (INF2820 — Data Abstraction)

(funcall #'equal 'eq 'equal) → nil

Stephan Oepen

Universitetet i Oslo & CSLI Stanford

oe@ifi.uio.no

Abstract Data Types

- `defstruct()` creates a new *abstract data type*, encapsulating a structure:

```
? (defstruct rule
    lhs rhs)
→ RULE
```

- `defstruct()` defines a new *constructor*, *accessors*, and a type *predicate*:

```
? (setf *foo* (make-rule :lhs 'S :rhs '(NP PP)))
→ #S(RULE :LHS S :RHS (NP PP))
? (listp *foo*) → nil
? (rule-p *foo*) → t
? (setf (rule-rhs *foo*) '(NP VP)) → (NP VP)
? *foo* → #S(RULE :LHS S :RHS (NP VP))
```

- abstract data types *encapsulate* a group of related data (i.e. an 'object').

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))`

Vectors and Arrays

- Multidimensional 'grids' of data can be represented as *vectors* or *arrays*;
- `(make-array (`*rank*$_1$ ... *rank*$_n$`))` creates an array with *n* dimensions;

```
? (setf *foo* (make-array '(2 5) :initial-element 0))
→ #((0 0 0 0 0) (0 0 0 0 0))
? (setf (aref *foo* 1 2) 42) → 42
```

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	42	0	0

- all dimensions count from zero; `aref()` accesses one individual cell;
- one-dimensional arrays are called *vectors* (abstractly similar to lists).