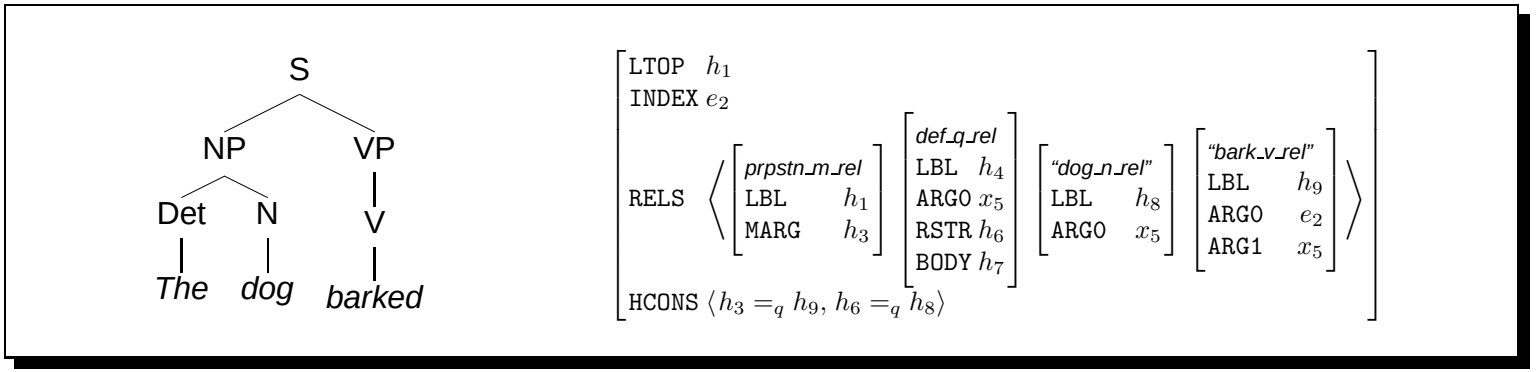

Algorithms for AI and NLP (INF4820 — Data Abstraction)

(defstruct fsa vocabulary table final)

Stephan Oepen and Jan Tore Lønning

Universitetet i Oslo

{ oe | jtl }@ifi.uio.no

Long Overdue: 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)` → `43`

  ? `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 $variable_{i+1}$.

Property Lists as Database Tuples

- A *property list* (or plist) represents any number of *key* – *value* pairings:

```
? (setf *foo* (list :artist "Dixie Chicks" :title "Home"))
  → (:ARTIST "Dixie Chicks" :TITLE "Home")
? (getf *foo* :title) → "Home"
? (getf *foo* :producer) → nil
? (setf (getf *foo* :title) "Fly") → "Fly"
? *foo* → (:ARTIST "Dixie Chicks" :TITLE "Fly")
```

- plists are regular lists, but always have an even number of elements;
- symbols whose name starts with a colon (‘:’) called *keyword symbols*;
- by convention, mostly use keyword symbols as names of keys in plists;
- but any Lisp object would be legitimate to use for plist keys or values.

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).

Abstract Data Types

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

```
? (defstruct cd
    artist title)
→ CD
```

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

```
? (setf *foo* (make-cd :artist "Dixie Chicks" :title "Home"))
  → #S(CD :ARTIST "Dixie Chicks" :TITLE "Home")
? (listp *foo*) → nil
? (cd-p *foo*) → t
? (setf (cd-title *foo*) "Fly") → "Fly"
? *foo* → #S(CD :ARTIST "Dixie Chicks" :TITLE "Fly")
```

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

Input and Output — Side Effects

- Input and output, to files or the terminal, is mediated through *streams*;
- the symbol `t` can be used to refer to the default stream, the terminal:

```
? (format t "line: ~a; token ‘~a’.~%" 42 "foo")
⇝ line: 42; token ‘foo’.
⟶ nil
```

- `(read` *stream* `nil)` reads one well-formed s-expression from *stream*;
- `(read-line` *stream* `nil)` reads one line of text, returning it as a string;
- the second argument to reader functions asks to return `nil` on end-of-file.

```
(with-open-file (stream "sample.txt" :direction :input)
  (loop
      for line = (read-line stream nil)
      while line do (format t "~a~%" line)))
```