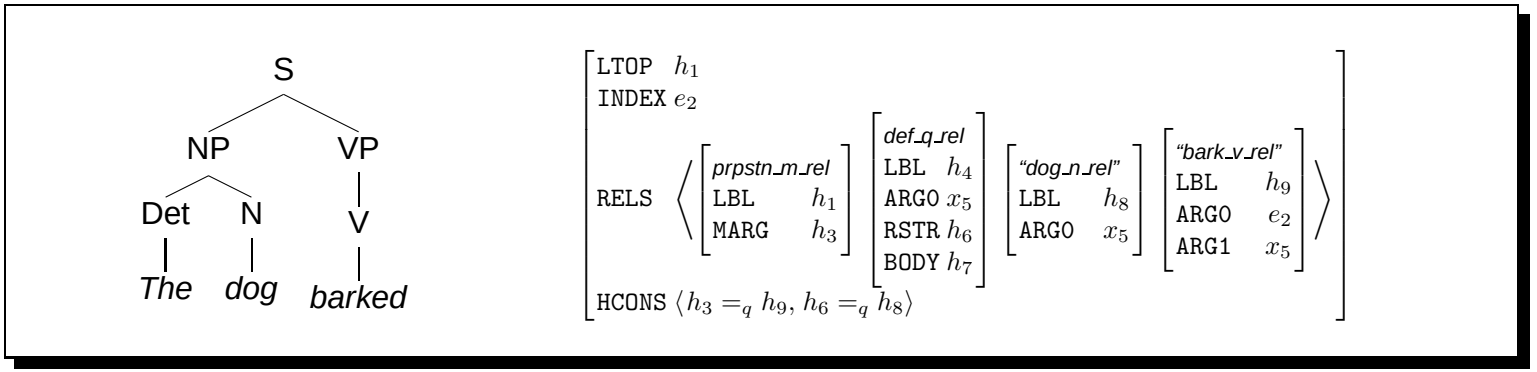


Algorithms for AI and NLP (INF4820 — Lisp & FSAs)

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

Stephan Oepen and Jonathon Read

Universitetet i Oslo

{ oe | jread }@ifi.uio.no

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).

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 specialized `loop()` directives:

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

Some Remarks on Choosing Among Container Types

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)))
```

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 $variable_{i+1}$.

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').

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: Finite-State Automata

/baa+!/

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

Pattern Matching: Finite-State Automata

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;

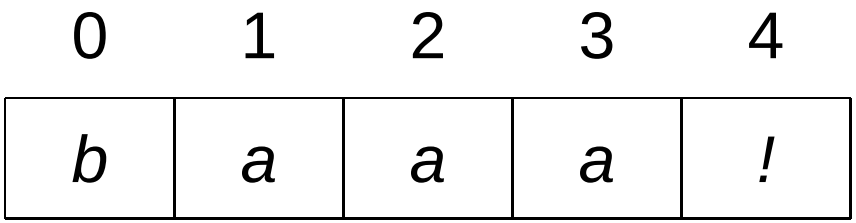
→ *accept* utterance when no more input, in a 'final' state; else *reject*.

Tracing the Recognition of a Simple Input

/baa+!/

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

Input Tape



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 $\longrightarrow$ 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)

```
 1  procedure nd-recognize(tape , fsa) ≡
 2    agenda ← {⟨0, 0⟩};
 3    do
 4      current ← pop(agenda);
 5      state ← first(current);
 6      index ← second(current);
 7      if (index = length(tape) and state is final state) then
 8        return accept;
 9      fi
10      for(next in fsa.transitions[state, tape[index]]) do
11        agenda ← agenda ∪ {⟨next, index + 1⟩}
12      od
13      if agenda is empty then return reject; fi
14    od
15  end
```