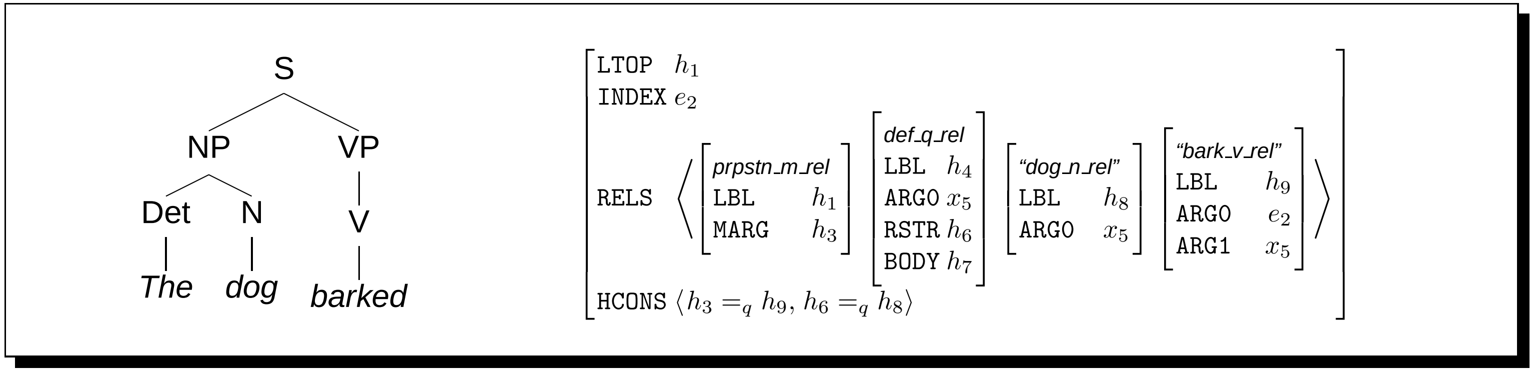


Algorithms for AI and NLP (INF4820 — Welcome)

(defun ! (n) (if (equal n 0) 1 (* n (! (- n 1)))))

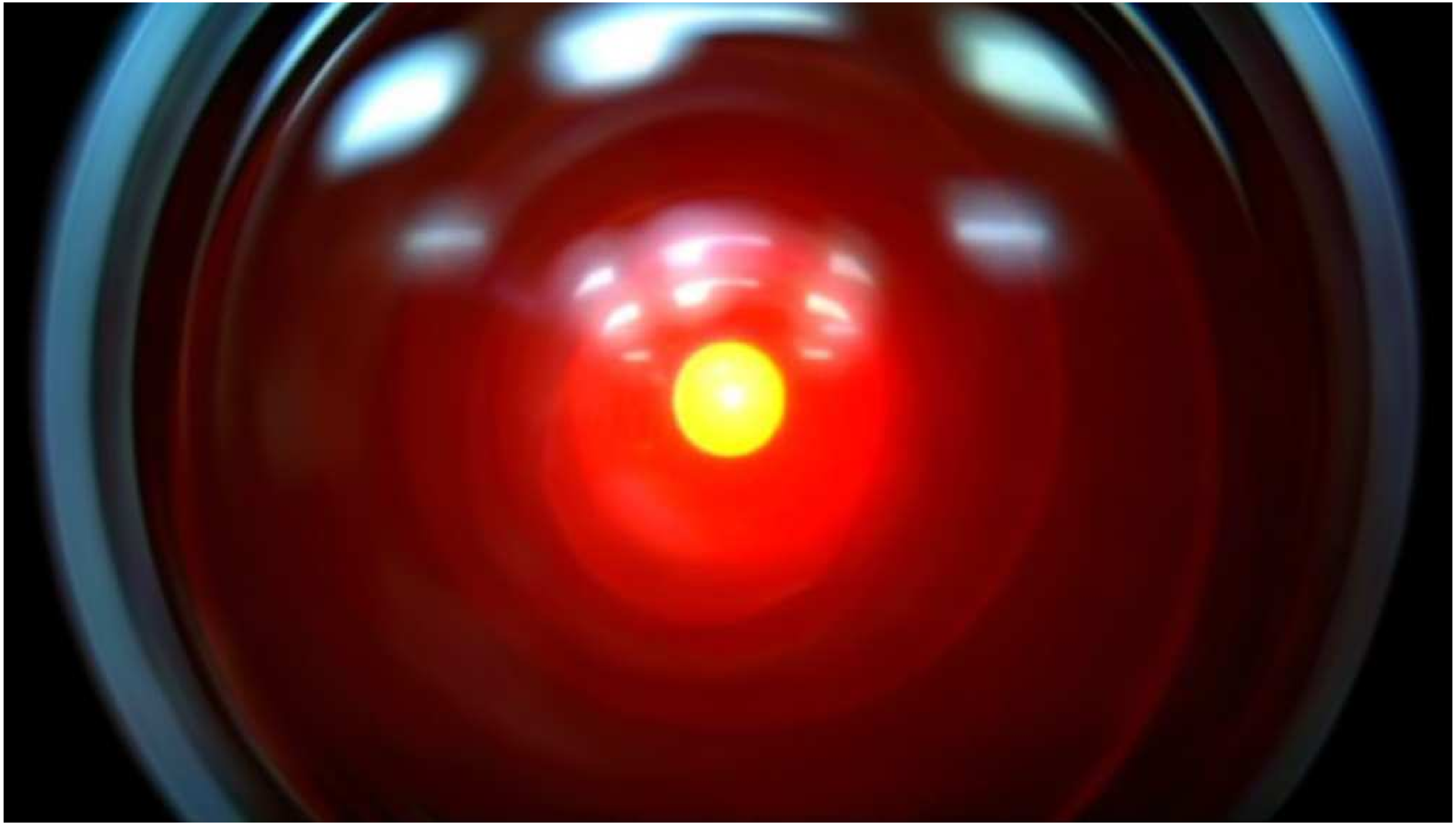
Stephan Oepen and Jonathon Read

Universitetet i Oslo

{ oe | jread }@ifi.uio.no

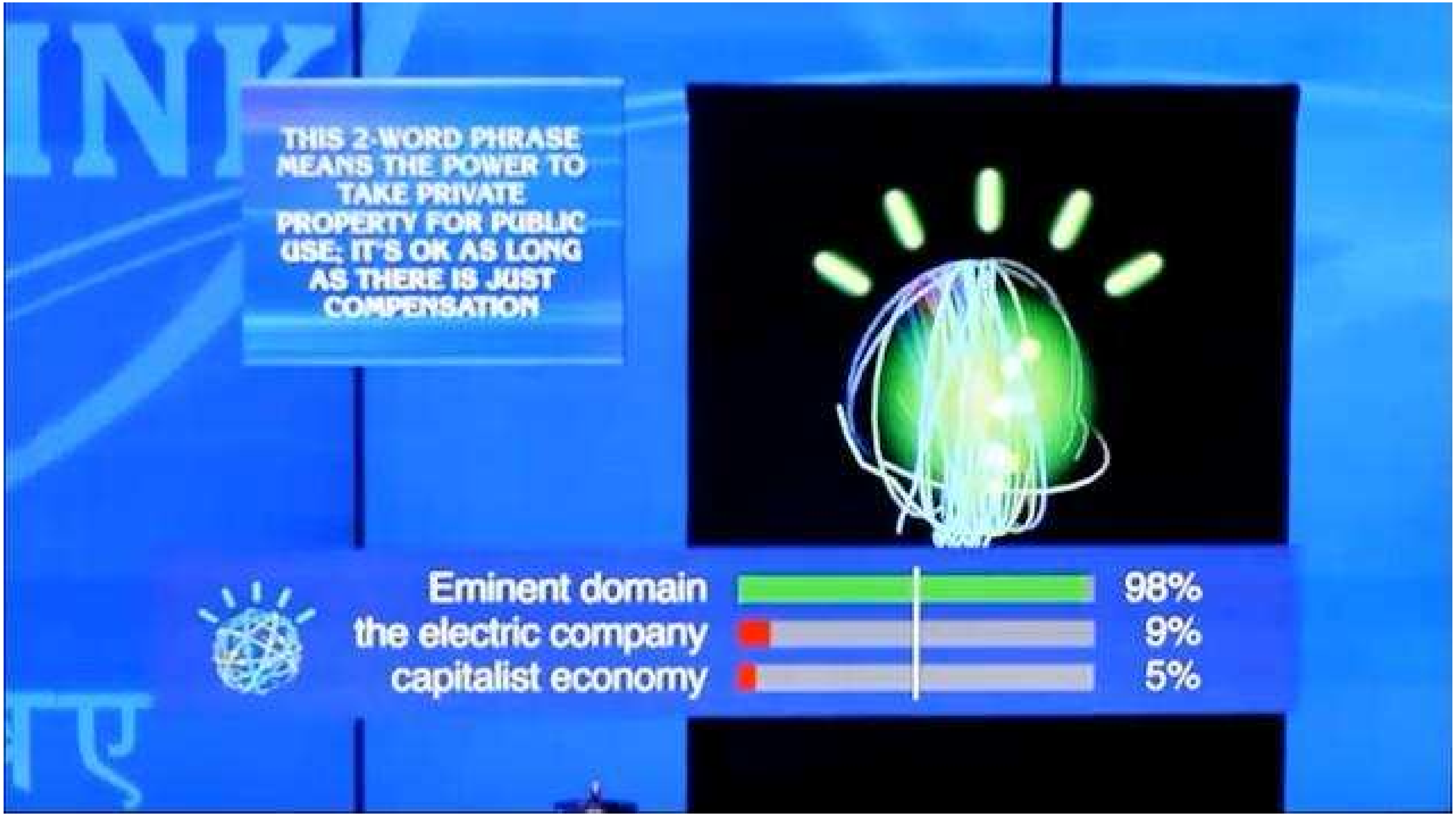
So, What Actually is AI and NLP

So, What Actually is AI and NLP



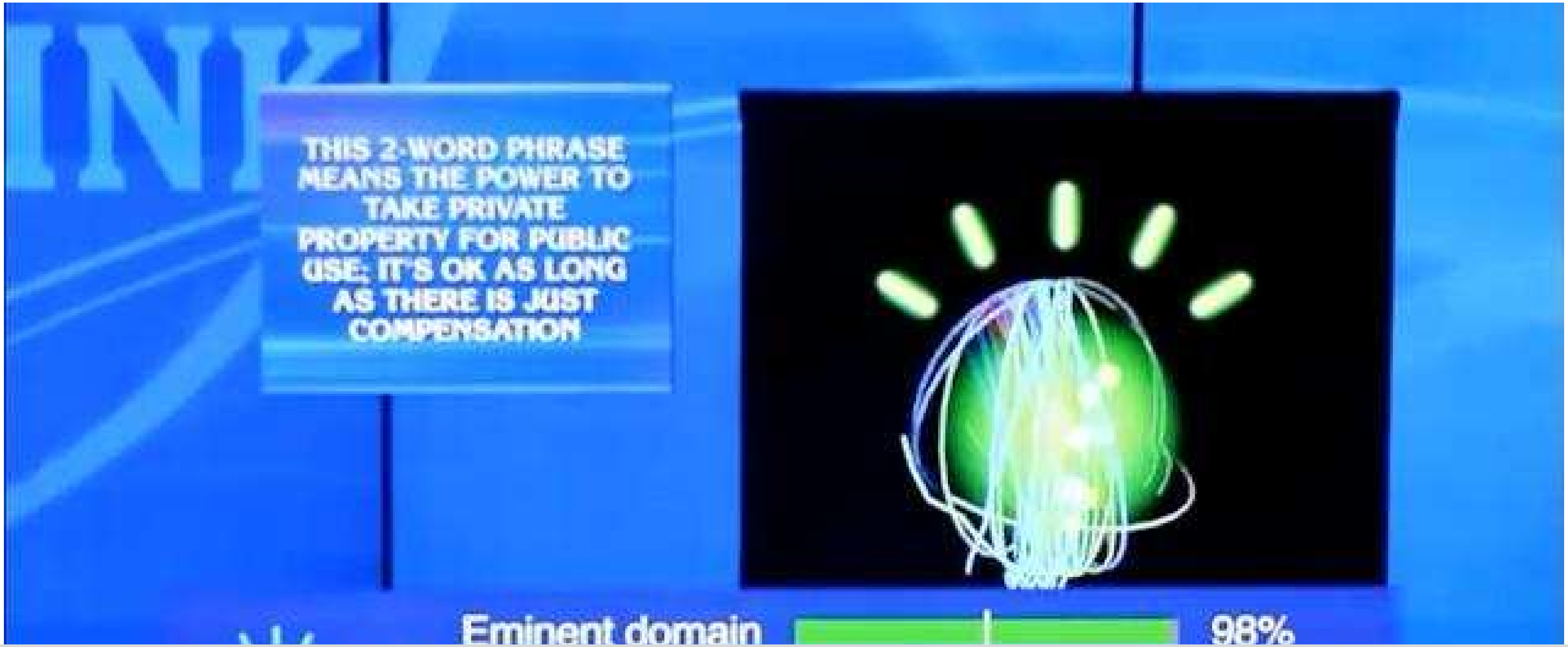
(2001: A Space Odyssey; HAL 9000; 1968)

So, What Actually is AI and NLP



(IBM Watson beats long-time *Jeopardy!* champions; 2011)

So, What Actually is AI and NLP



→ (young) interdisciplinary science: language, computing, cognition;

→ (again) culturally and commercially relevant for 'knowledge society'.

What Makes Natural Language a Hard Problem?

```
< Den andre veien mot Bergen er kort. --- 12 x 30 x 25 = 25
> The other path towards Bergen is short. {0.58} (1:1:0).
> The other road towards Bergen is short. {0.56} (1:0:0).
> The second road towards Bergen is short. {0.55} (2:0:0).
> That other path towards Bergen is a card. {0.54} (0:1:0).
> That other road towards Bergen is a card. {0.54} (0:0:0).
> The second path towards Bergen is short. {0.51} (2:1:0).
> The other road against Bergen is short. {0.48} (1:2:0).
> The second road against Bergen is short. {0.48} (2:2:0).
  ...
> Short is the other street towards Bergen. {0.33} (1:4:0).
> Short is the second street towards Bergen. {0.33} (2:4:0).
  ...
```

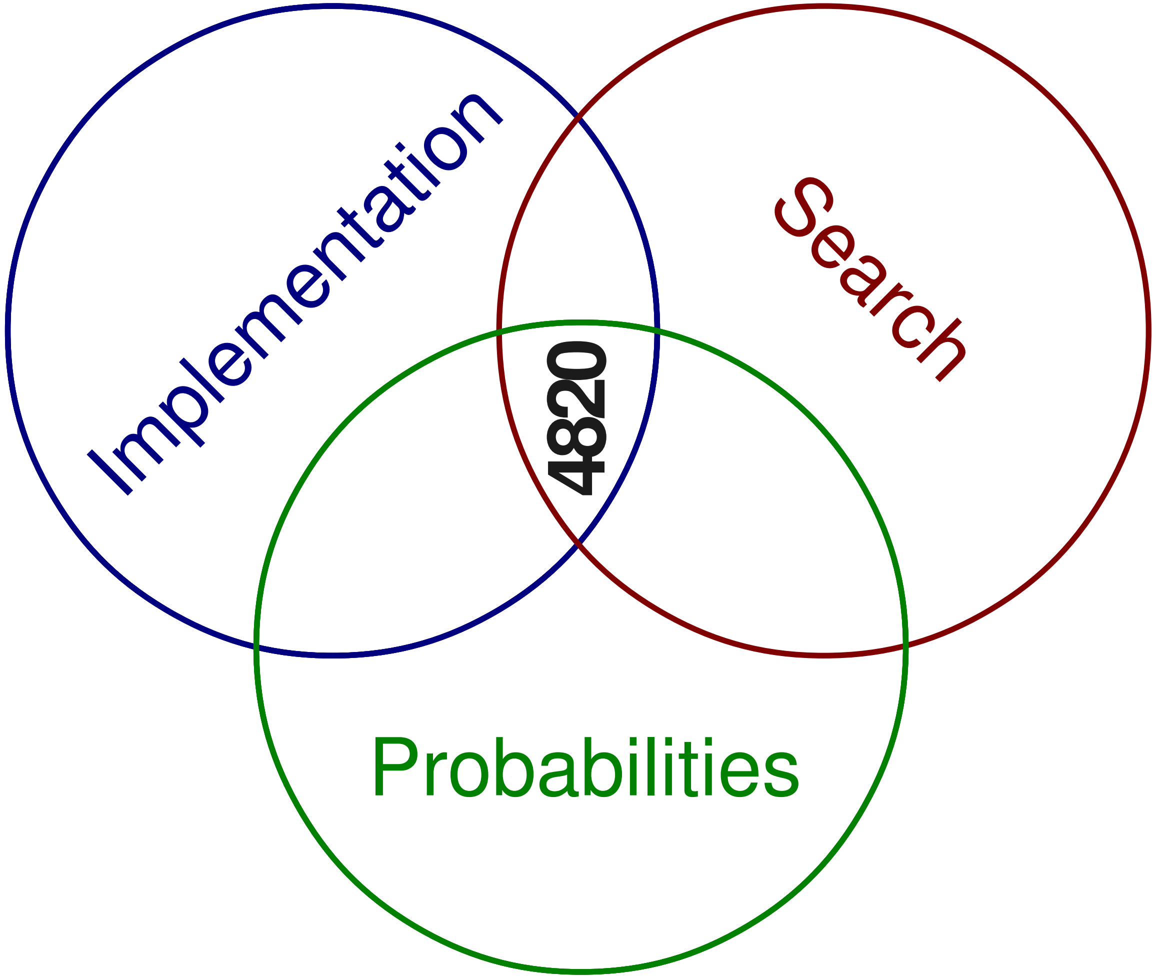
What Makes Natural Language a Hard Problem?

```
< Den andre veien mot Bergen er kort. --- 12 x 30 x 25 = 25
> The other path towards Bergen is short. {0.58} (1:1:0).
> The other road towards Bergen is short. {0.56} (1:0:0).
> The second road towards Bergen is short. {0.55} (2:0:0).
> That other path towards Bergen is a card. {0.54} (0:1:0).
> That other road towards Bergen is a card. {0.54} (0:0:0).
> The second path towards Bergen is short. {0.51} (2:1:0).
> Th … ).
> Th … )).
  ...
> Sh … 0).
> Sh … :0).
  ...
```

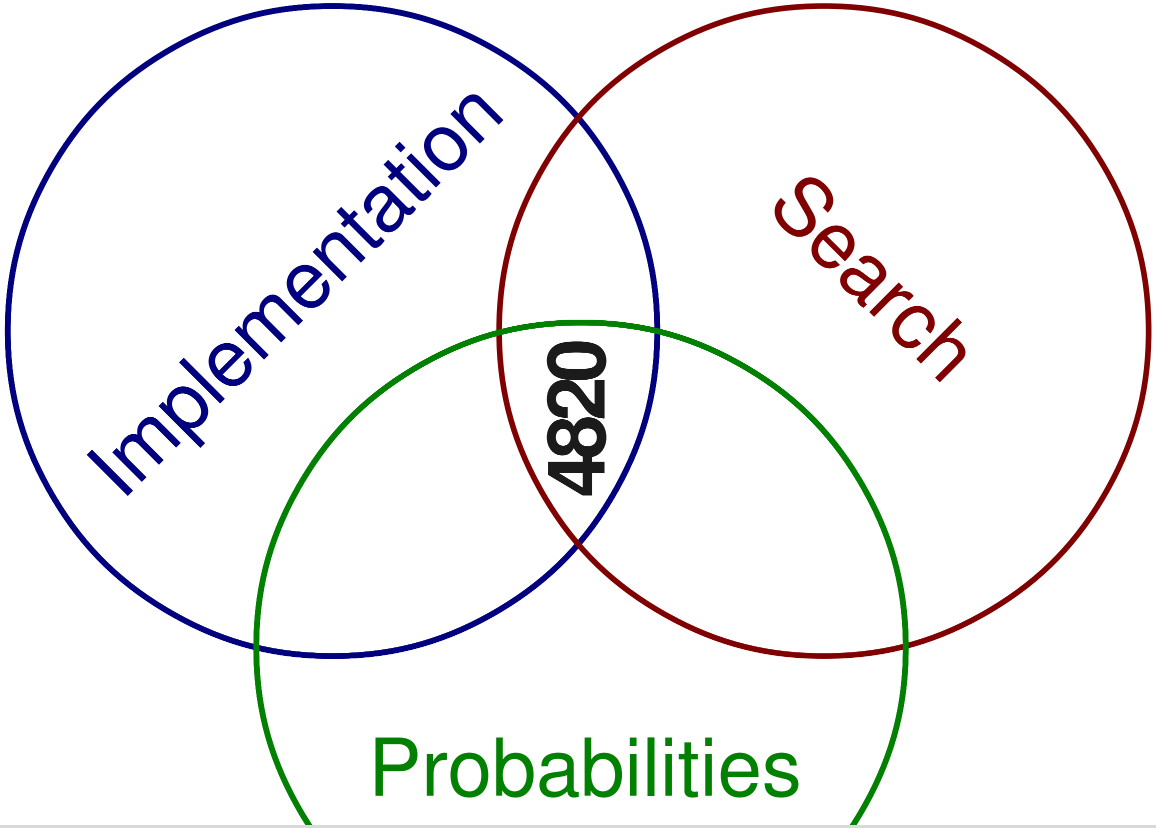
Scraped Off the Internet

```
The other way to Bergen is short.
the road to the other bergen is short .
Den other roads against Boron Gene are short.
Other one autobahn against Mountains am abrupt.
```

INF4820: A Very High-Level Perspective



INF4820: A Very High-Level Perspective



Efficient and Scalable Algorithms and Data Structures for Searching (Probabilistically) Weighted Solution Spaces

Well, Who is Actually Working on This?

In the next three to five years, [...] mobile devices [...] will become prevalent. [...] Desired technologies will soon replace menus and graphic user interfaces with natural-language interfaces. — People so much want to speak English to their computer. (Steve Ballmer, December 2005)

Well, Who is Actually Working on This?

In the next three to five years, [...] mobile devices [...] will become prevalent. [...] Desired technologies will soon replace menus and graphic user interfaces with natural-language interfaces. — People so much want to speak English to their computer. (Steve Ballmer, December 2005)

IBM has unveiled the details of its plans to build a computing system that can understand complex questions and answer with enough precision and speed to compete on America's favorite quiz show, Jeopardy!. (IBM Press Release, April 27, 2009)

Families of Language Processing Tasks

Speech Recognition and Synthesis

Summarization & Text Simplification

(High Quality) Machine Translation

Information Extraction — Text Understanding

Grammar & Controlled Language Checking

Natural Language Dialogue Systems

Families of Language Processing Tasks

Speech Recognition and Synthesis

Summarization & Text Simplification

(High Quality) Machine Translation

Information Extraction — Text Understanding

Grammar & Controlled Language Checking

Natural Language Dialogue Systems

Families of Language Processing Tasks

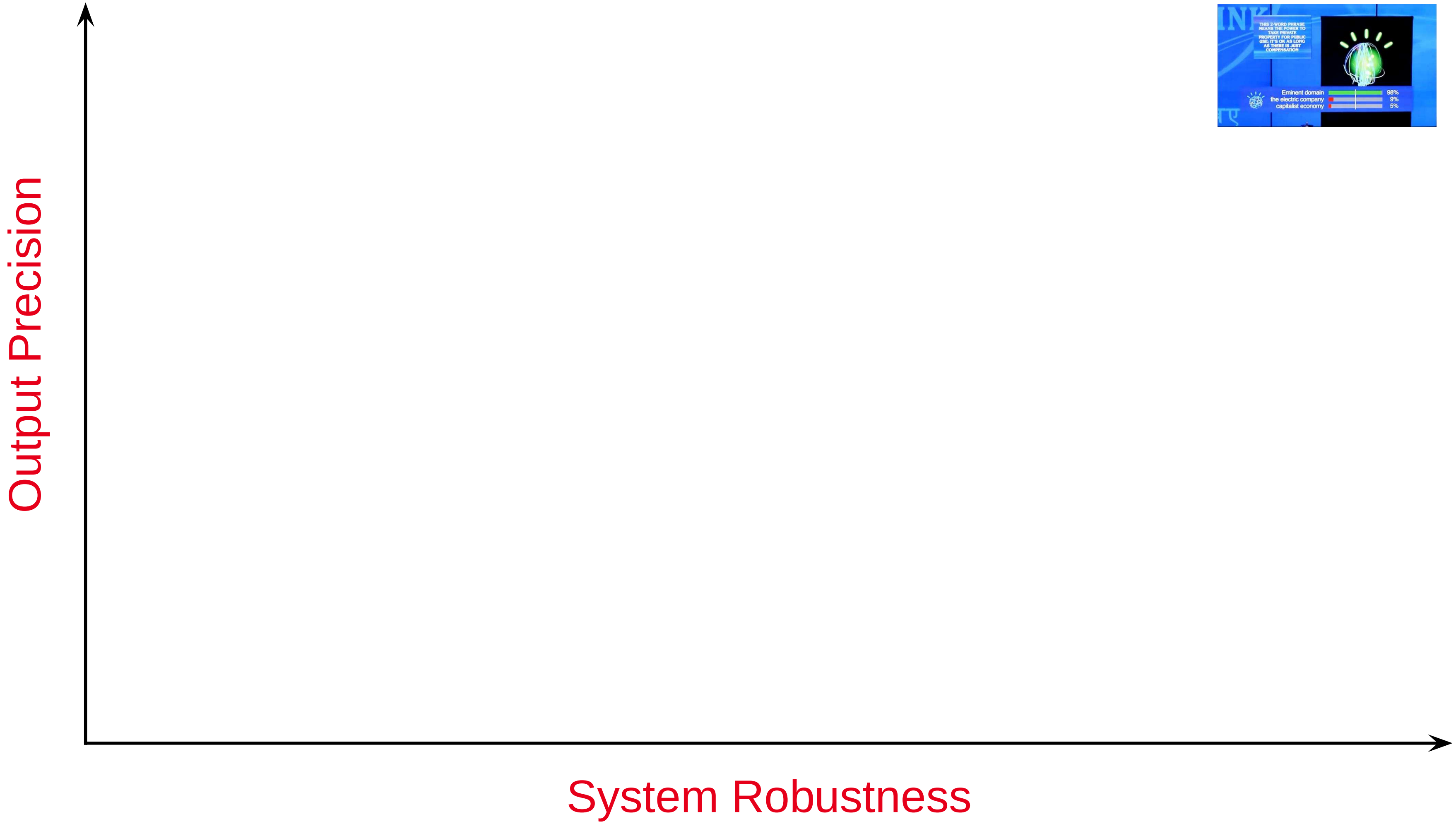
- Speech Recognition and Synthesis
- Summarization & Text Simplification
- (High Quality) Machine Translation
- Information Extraction — Text Understanding
- Grammar & Cor guage Checking
- Natural Lang gue Systems

Focus on Accuracy

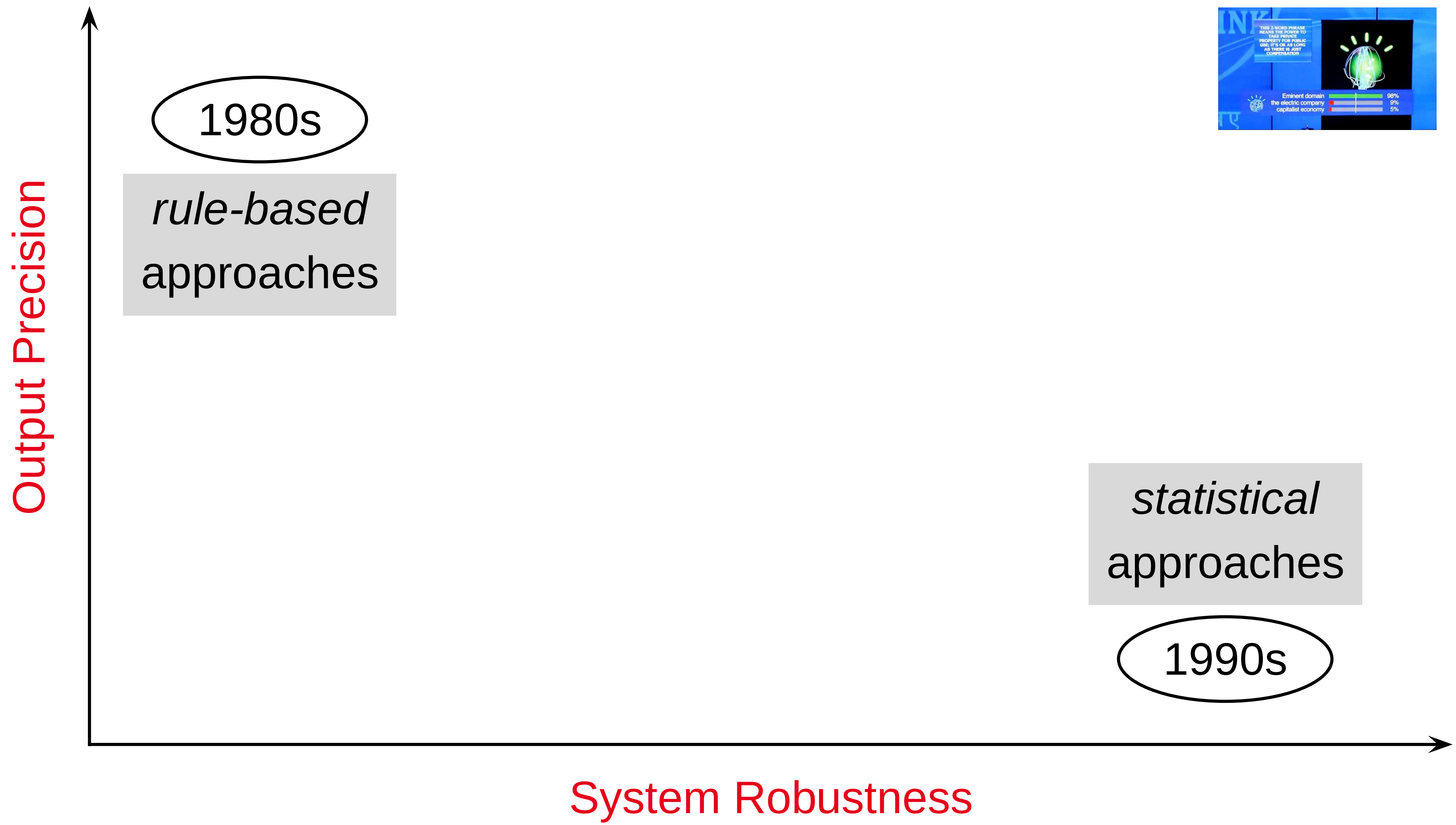
Use of Semantics

(Mostly) English

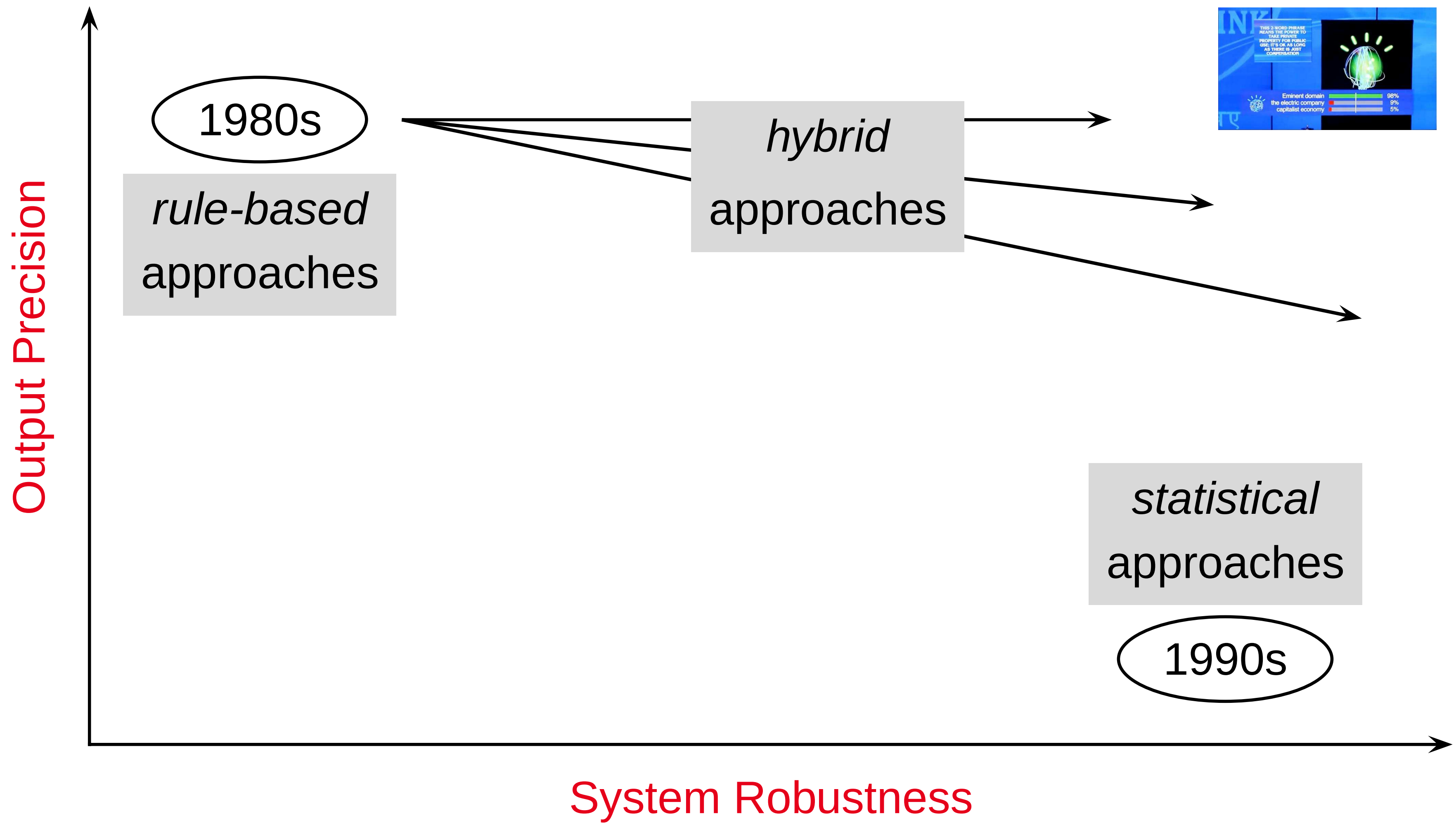
The Holy Grail: Balancing Robustness and Precision



The Holy Grail: Balancing Robustness and Precision



The Holy Grail: Balancing Robustness and Precision



INF4880 — What We Are About to Do (and Why)

Comments on Course & Background Literature

Why Common-Lisp for (Symbolic) Programming?

- Arguably most widely used language for ‘symbolic’ computation;
- easy to learn: extremely simple syntax; straightforward semantics;
- a rich language: multitude of built-in data types and operations;
- full standardization; Common-Lisp has been stable for a decade;
- *Ruby was a Lisp originally, in theory.* [Yukihiro Matsumoto; 2006];

→ for our purposes, (at least) as good a choice as any other language.

$$n! \equiv \begin{cases} 1 & \text{for } n = 0 \\ n \times (n-1)! & \text{for } n > 0 \end{cases}$$

```
(defun ! (n)
  (if (equal n 0)
    1
    (* n (! (- n 1)))))
```

Why Common-Lisp for (Symbolic) Programming?

- Arguably most widely used language for ‘symbolic’ computation;
- easy to learn: extremely simple syntax; straightforward semantics;
- a rich language: multitude of built-in data types and operations;
- full standardization; Common-Lisp has been stable for a decade;
- *Ruby was a Lisp originally, in theory.* [Yukihiro Matsumoto; 2006];

→ for our purposes, (at least) as good a choice as any other language.

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use Lisp itself a lot.
[Eric Raymond, 2001]

Common-Lisp: Syntax

- Numbers: `42`, `3.1415`, `1/3`;
- strings: `"foo"`, `"42"`, `"(bar)"`;
- symbols: `pi`, `t`, `nil`, `foo`, `FoO`;
- lists: `(1 2 3 4 5)`, `()`, `nil`,

```
(defun ! (n)
  (if (equal n 0)
    1
    (* n (! (- n 1)))))
```

- Lisp manipulates *symbolic expressions* (known as ‘sexps’);
- sexps come in two fundamental flavours, atoms and lists;
- atoms include numbers, strings, symbols, structures, et al.;
- sexps are used to represent *both* program data and program code;
- rather few ‘magic’ characters: `‘(’`, `‘)’`, `‘"’`, `‘’’`, `‘;’`, `‘#’`, `‘|’`, `‘‘’`;
- all operators use *prefix* notation;
- symbol case does *not* matter.

Common-Lisp: Semantics (aka Evaluation)

- Constants (e.g. numbers and strings, `t` and `nil`) evaluate to themselves:

  ? `3.1415` → `3.1415`   ? `"foo"` → `"foo"`   ? `t` → `t`   ? `nil` → `nil`

- symbols evaluate to their associated value (if any):

  ? `pi` → `3.141592653589793`

  ? `foo` → *error* (unless a value was assigned earlier)

- lists are function calls; the first element is interpreted as an operator and invoked with the *values* of all remaining elements as its arguments:

  ? `(* pi (+ 2 2))` → `12.566370614359172`;

- the `quote()` operator (abbreviated as '’') suppresses evaluation:

  ? `(quote (+ 2 2))` → `(+ 2 2)`

  ? `'foo` → `foo`

A Couple of List Operations

- `first()` and `rest()` destructure lists; `cons()` builds up new lists:

  ? `(first '(1 2 3))` → `1`

  ? `(rest '(1 2 3))` → `(2 3)`

  ? `(first (rest '(1 2 3)))` → `2`

  ? `(rest (rest (rest '(1 2 3))))` → `nil`

  ? `(cons 0 '(1 2 3))))` → `(0 1 2 3)`

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

- many additional list operations (derivable from the above primitives):

  ? `(list 1 2 3)` → `(1 2 3)`

  ? `(append '(1 2 3) '(4 5 6))` → `(1 2 3 4 5 6)`

  ? `(length '(1 2 3))` → `3`

  ? `(reverse '(1 2 3))` → `(3 2 1)`

Assigning Values — 'Generalized Variables'

- `defparameter()` declares a 'global variable' and assigns a value:

  ? `(defparameter *foo* 42)` → `*FOO*`

  ? `*foo*` → `42`

- `setf()` associates ('assigns') a value to a symbol (a 'variable'):

  ? `(setf *foo* (+ *foo* 1))` → `43`

  ? `*foo*` → `43`

  ? `(setf *foo* '(1 1 3))` → `(1 1 3)`

- `setf()` can also alter the values associated to 'generalized variables':

  ? `(setf (first (rest *foo*)) 2)` → `2`

  ? `*foo*` → `(1 2 3)`

  ? `(setf (cons 0 *foo*) 2)` → *error*

Predicates — Conditional Evaluation

- A *predicate* tests some condition and evaluates to a boolean truth value; `nil` indicates *false* — anything non-`nil` (including `t`) indicates *true*:

```
? (listp '(1 2 3)) → t
? (null (rest '(1 2 3))) → nil
? (or (not (numberp *foo*)) (and (>= *foo* 0) (< *foo* 42)))
  → t
? (equal (cons 1 (cons 2 (cons 3 nil))) '(1 2 3)) → t
? (eq (cons 1 (cons 2 (cons 3 nil))) '(1 2 3)) → nil
```

- conditional evaluation proceeds according to a boolean truth condition:

```
? (if (numberp *foo*)
    (+ *foo* 42)
    (first (rest *foo*)))
  → 2
```

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;
- a function evaluates to the value of the *last* sexp in the function *body*.