Logic and Knowledge Representation

Language Processing, Meta-programming

1 June 2018

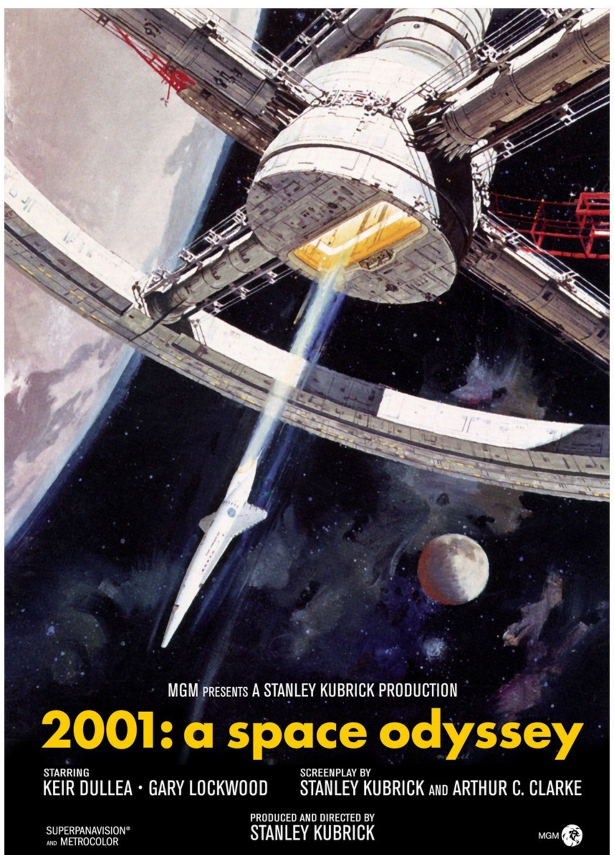
Giovanni Sileno gsileno@enst.fr

Télécom ParisTech, Paris-Dauphine University

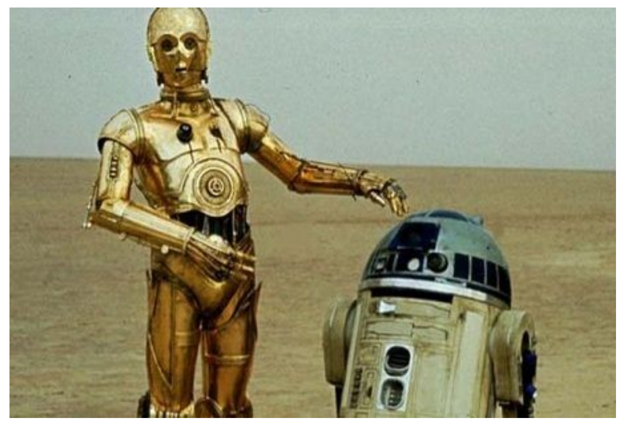
Natural Language Processing

About talking machines…

- The dream of machine talking to humans is present in many fictional works…

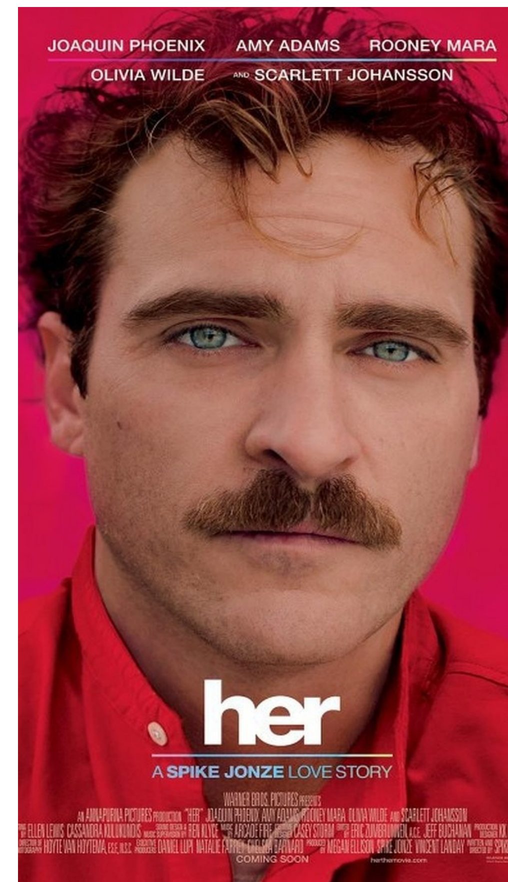


Hal 9000



C-3PO, R2-D2 (Star Wars)

Data (Star Trek)



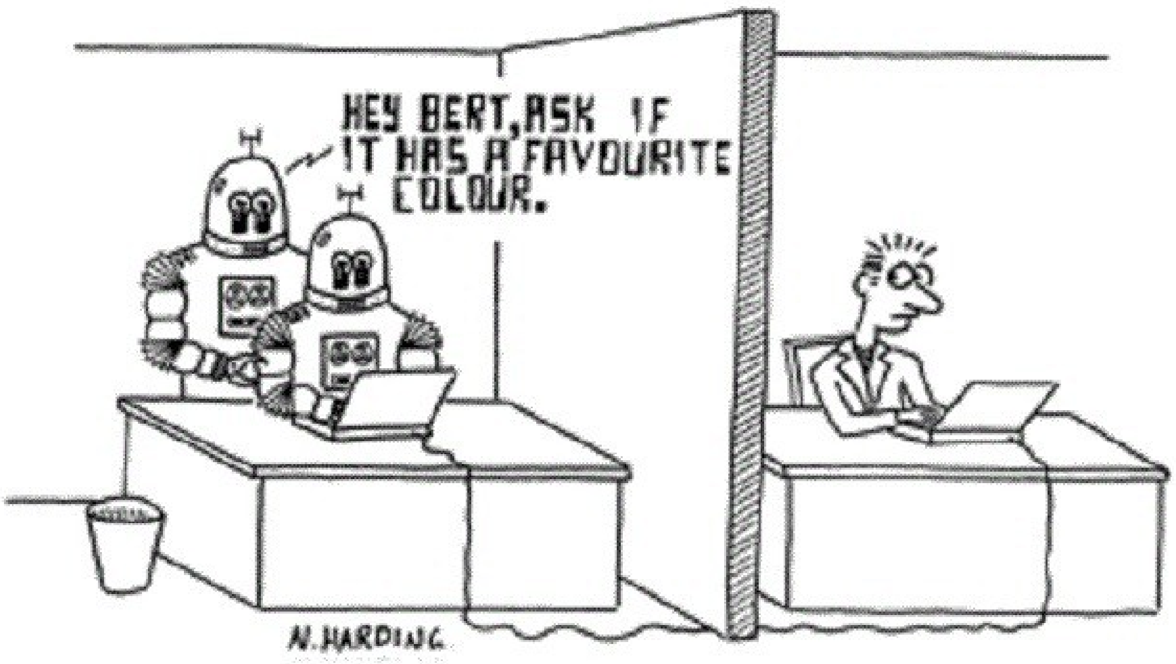
Samantha

About talking machines…

- The dream of machine talking to humans is present in many fictional works… even in ancient times!

From the Iliad, Book XVIII:
"There were golden handmaids also who worked for him [Hephaestus], and were like real young women, with sense and reason, voice also and strength, and all the learning of the immortals."

About talking machines…

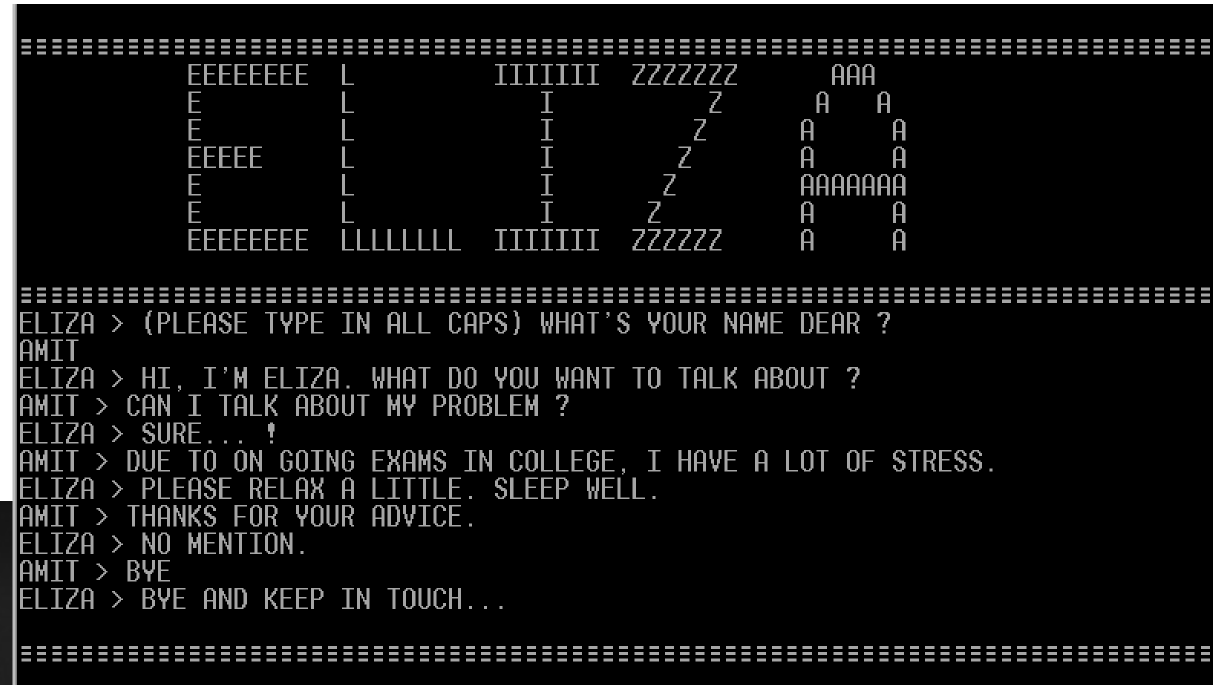


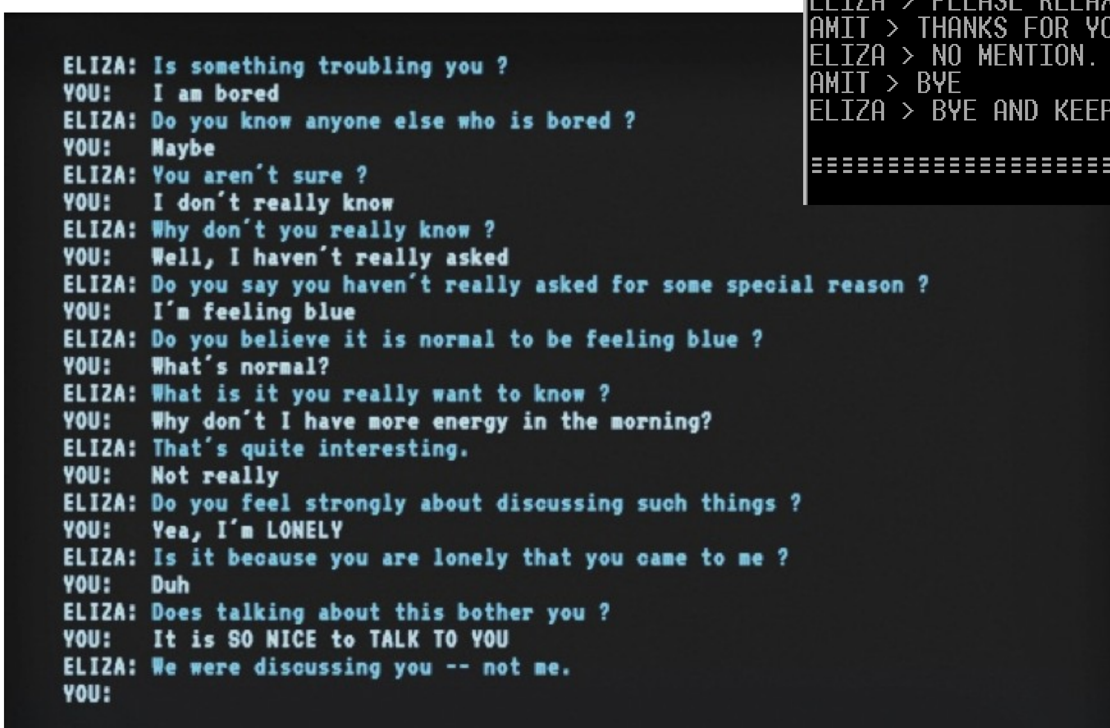
Variations of the (in)famous "Turing test"

First “concrete” attempts

ELIZA

Weizenbaum ~1965

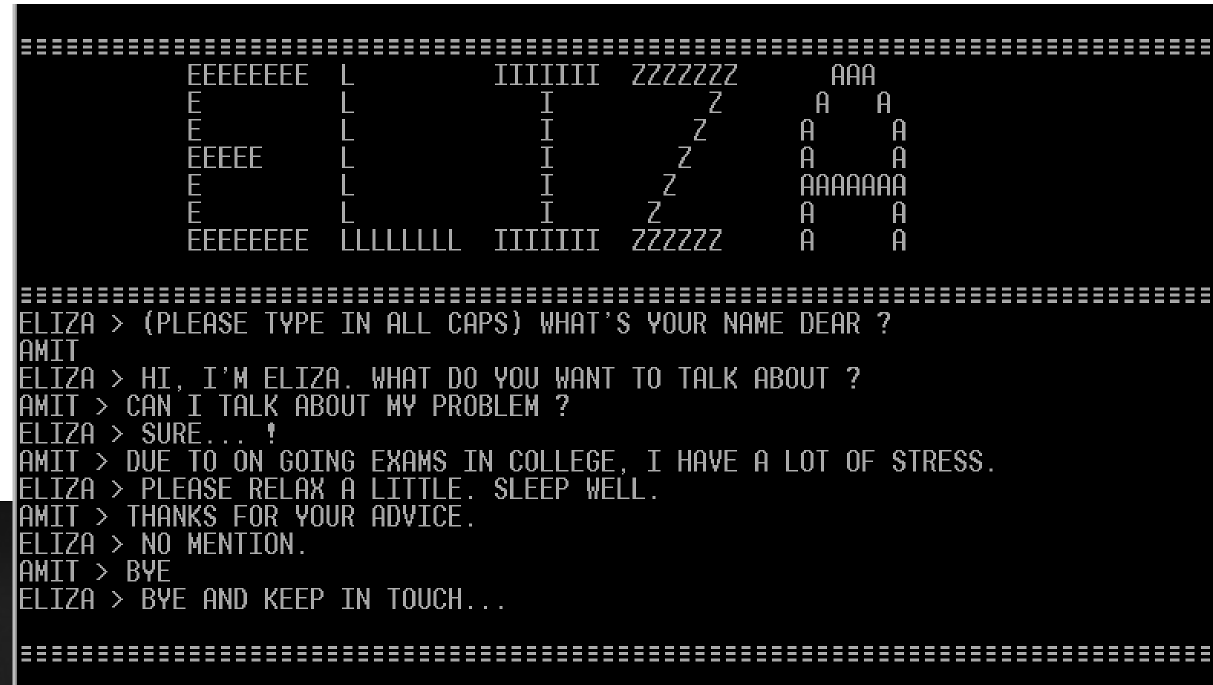


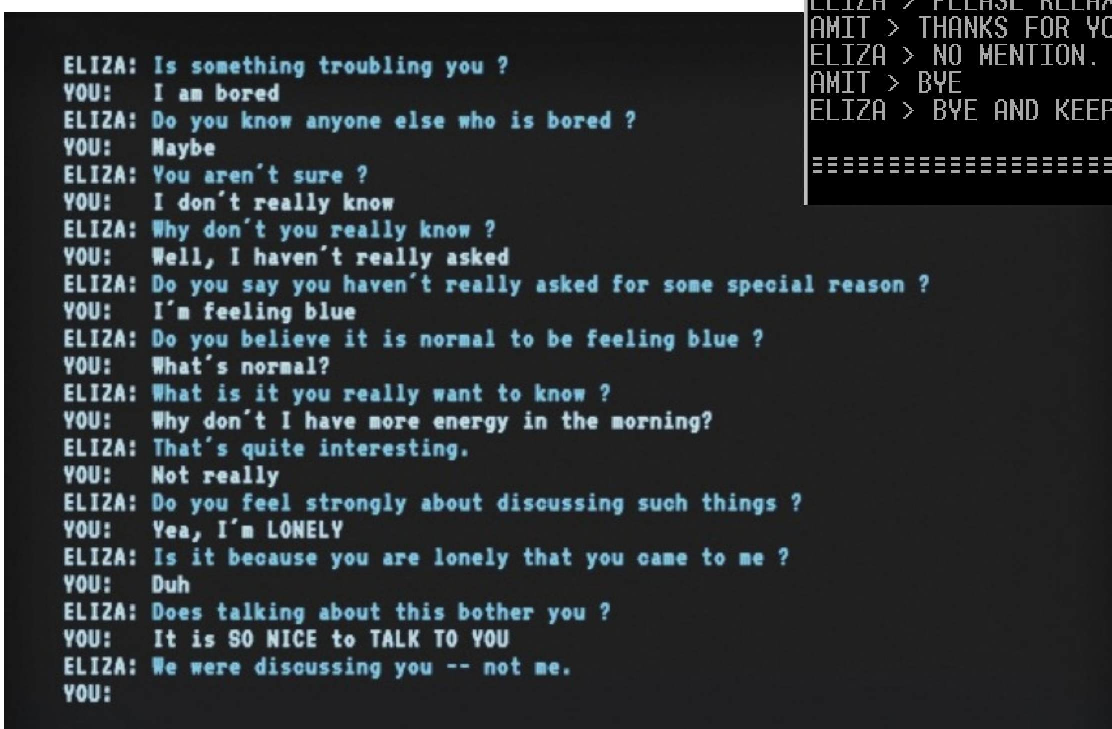


First “concrete” attempts

ELIZA

Weizenbaum ~1965



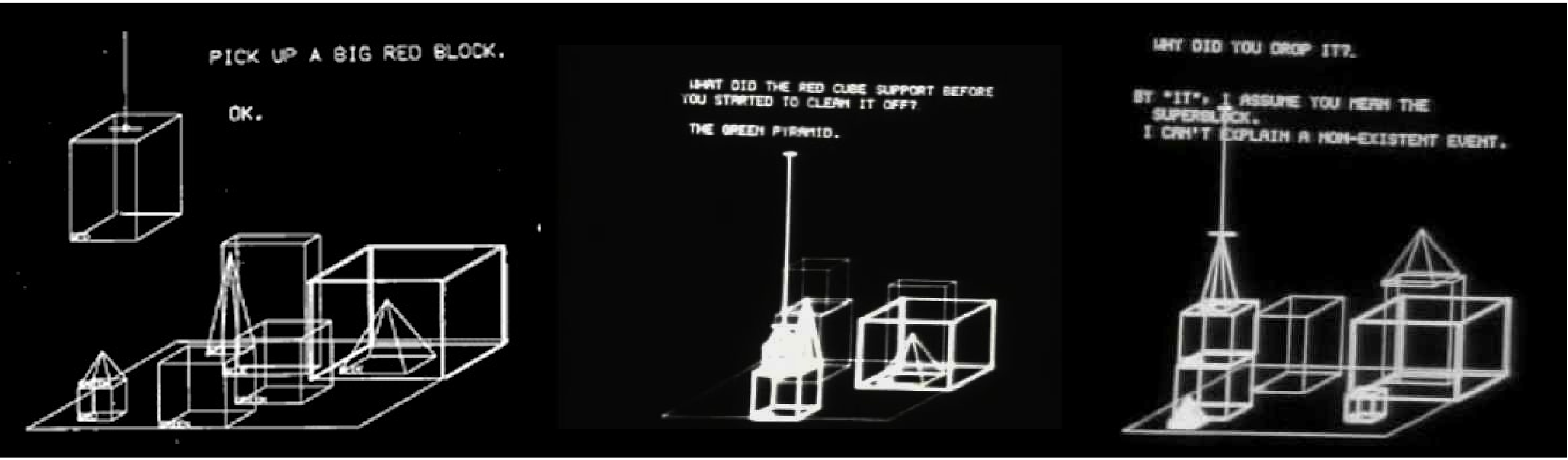


If you have emacs,
try `M-x doctor`

First “concrete” attempts

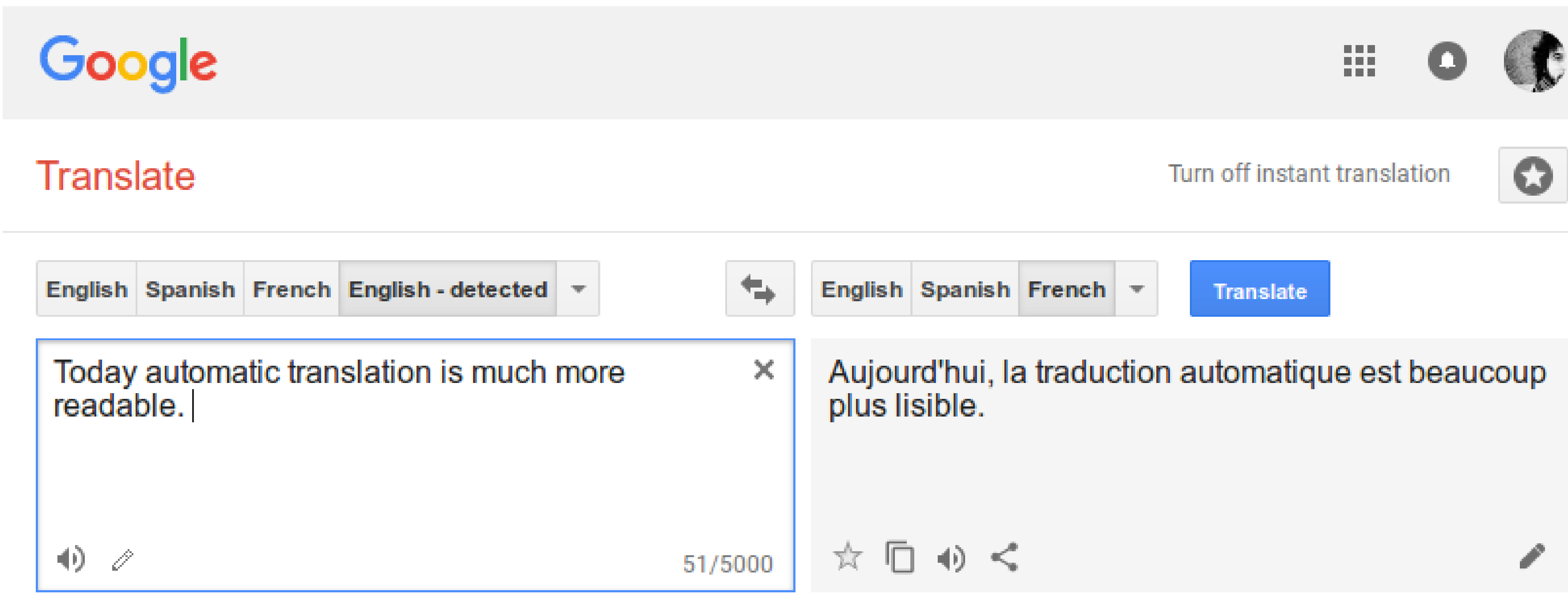
SHRDLU

Winograd ~1969



- Deeper understanding
- but limited to a simple *blocks* world

Today?



- But do automatic translators “understand” what we say?

Winograd Schema Challenge

- Proposed by Levesque in 2014 to go beyond the Turing test, it counts today 140 sentences as:

"The city councilmen refused the demonstrators a permit because they [feared/advocated] violence."

- *To whom they refers?*

Winograd Schema Challenge

- Proposed by Levesque in 2014 to go beyond the Turing test, it counts today 140 sentences as:

> "The city councilmen refused the demonstrators a permit because they [feared/advocated] violence."

- *To whom they refers?*
- Problem: resolving **anaphoras**

/ai θot.../

(Phonology, the study of pronunciation)

go/going

(Morphology, the study of word constituents)

I thought they're never going to hear me 'cause they're screaming all the time. [Elvis Presley]

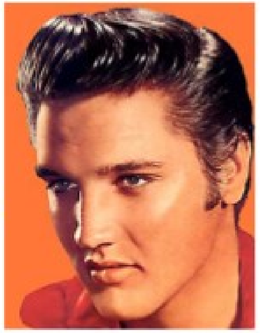
Sentence

Noun phrase

Verbal phrase

(Syntax, the study of grammar)

"I" =



It doesn't matter what I sing.

(Pragmatics, the study of language use)

(Semantics, the study of meaning)

[Slide by Fabian Suchanek, Télécom ParisTech]

/ai θot.../

(Phonology, the study of pronunciation)

go/going

(Morphology, the study of word constituents)

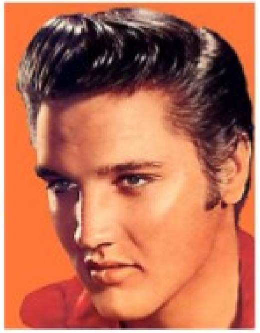
I thought they're never going to hear me 'cause they're screaming all the time. [Elvis Presley]

Sentence

Noun phrase | Verbal phrase

(Syntax, the study of grammar)

"I" =



It doesn't matter what I sing.

(Pragmatics, the study of language use)

(Semantics, the study of meaning)

- All these levels play a role with language!

not only in verbal language...

Vittore Carpaccio,
Due Dame, ~1495

not only in verbal language...

Beware of context!

Vittore Carpaccio,
Due Dame + Caccia in valle, ~1495
reconstruction of the original painting

Language Processing in Prolog

Prolog and Context-Free Grammars

- Alain Colmerauer and Philippe Roussel conceived Prolog (1972) to facilitate syntaxic processing, following the theory of ***context-free grammars***.
- CFGs were introduced in linguistics by Noam Chomsky to clearly distinguish syntax from semantics [and to attack simple statistical models of language.]

Colorless green ideas sleep furiously.

Furiously sleep ideas green colorless.

Context-Free Grammar

- A **context-free grammar** G is defined by

$$G = (V , \Sigma , R , S)$$

  - V is the finite set of **non-terminal** characters (variables), standing for the syntaxic category
  - Σ is a finite set of **terminal** symbols, disjoint from V, standing for the actual content of the sentence
  - R is a set of rewrite or **production rules** of the grammar, i.e. mappings from V to $(V \cup \Sigma)^*$ (* = Kleene star symbol)
  - S is the **start symbol**, used to represent the whole sentence (or program). It belongs to V.

Context-Free Grammar

- A **context–free grammar** G is defined by

$$G = (V , \Sigma , R , S)$$

- The **language** L(G) of a grammar G is defined as :

$$L(G) = \{ w \in \Sigma^* / S \Rightarrow^* w \}$$

- A **word** in L(G) derives from S and contains only terminal symbols.
- A language L is a context-free language if there is a context-free grammar G, such that L(G) = L.

Regular Expressions

- Regular expressions consist of:
  - constants, denoting sets of strings
    - ∅ denoting the empty set: {}
    - ε denoting the set containing only the empty string: {""}
    - a denoting the set containing only the string "a": {"a"}

Regular Expressions

- Regular expressions consist of:
  - constants, denoting sets of strings
    - ∅ denoting the empty set: {}
    - ε denoting the set containing only the empty string: {""}
    - a denoting the set containing only the string "a": {"a"}
  - operator symbols, denoting operations over sets.

    given two sets denoted with R and S, we have:
    - RS (**concatenation**): denotes the set of strings obtained by concatenanting a string of R and a string of S
    - R|S (**alternance**): denotes the set of strings obtained by the union of R and S
    - R* (**Kleene star**): denotes set the including ε, and all possible concatenations of strings in R (closed under concatenation).

Context-Free vs Regular languages

- A regular language is a language that can be expressed through a regular expression, or equivalently, by a ***finite state machine*** (Kleene's Theorem).
- All regular languages are context-free languages, but not otherwise.
- Example: $\{ 0^n 1^n : n \in N \}$ is not regular

Context-Free Grammar

- A CFG allows us to say whether a sentence is syntactically correct (***recogniser***) and what is their syntactic structure (***parser***).

Context-Free Grammar

- A CFG allows us to say whether a sentence is syntactically correct (***recogniser***) and what is their syntactic structure (***parser***).
- Example:

```
s ⇒ np vp
np ⇒ det n
vp ⇒ v np
vp ⇒ v
det ⇒ a
det ⇒ the
n ⇒ woman
n ⇒ man
v ⇒ kisses
```

```
  a man kisses a woman.
  a woman kisses a man.
  a man kisses a man.
  a woman kisses a woman.
* kisses woman.
* a man kisses woman.
? a man kisses.
```

CFG recognition in Prolog

- Prolog implementation using **difference lists:**

```
s ⇒ np vp              s(X, Z) :- np(X, Y), vp(Y, Z).
np ⇒ det n             np(X, Z) :- det(X, Y), n(Y, Z).
vp ⇒ v np              vp(X, Z) :- v(X, Y), np(Y, Z).
vp ⇒ v                 vp(X, Z) :- v(X, Z).
det ⇒ a                det([the|W], W).
det ⇒ the              det([a|W], W).
n ⇒ woman              n([woman|W], W).
n ⇒ man                n([man|W], W).
v ⇒ kisses             v([kisses|W], W).

                       ?- s([a,woman,kisses,a,man], []).
                          True.
```

From CFG to DCG

- Rewriting it as **definite clause grammars** (**DCG**):

s ⇒ **np vp**	`s --> np, vp.`
np ⇒ **det n**	`np --> det, n.`
vp ⇒ **v np**	`vp --> v, np.`
vp ⇒ **v**	`vp --> v.`
det ⇒ **a**	`det --> [the].`
det ⇒ **the**	`det --> [a].`
n ⇒ **woman**	`n --> [woman].`
n ⇒ **man**	`n --> [man].`
v ⇒ **kisses**	`v --> [kisses].`

```
?- s([a,woman,kisses,a,man], []).
   True.
```

A computational example

Backus–Naur Form (BNF):

<expr> ::= <num> | <num> + <expr> | <num> - <expr>

```
expr ⇒ num
expr ⇒ num + expr
expr ⇒ num – expr
num  ⇒ 0
num  ⇒ 1
num  ⇒ 2
num  ⇒ ...
```

A computational example

Backus–Naur Form (BNF):

<expr> ::= <num> | <num> + <expr> | <num> - <expr>

```
expr ⇒ num
expr ⇒ num + expr
expr ⇒ num – expr
num  ⇒ 0
num  ⇒ 1
num  ⇒ 2
num  ⇒ ...
```

```
expr --> num.
expr --> num, [+], expr.
expr --> num, [-], expr.
num --> [D], {number(D)}.

expr_recognize(L)
   :- expr(L, []).
```

A computational example

Backus–Naur Form (BNF):

<expr> ::= <num> | <num> + <expr> | <num> - <expr>

```
expr(Z) --> num(Z).
expr(Z) --> num(X), [+], expr(Y).
expr(Z) --> num(X), [-], expr(Y).
num(D) --> [D], {number(D)}.

expr_compute(L, V)
   :- expr(V, L, []).
```

- first, create space for a value to be passed

A computational example

Backus–Naur Form (BNF):

<expr> ::= <num> | <num> + <expr> | <num> - <expr>

```
expr(Z) --> num(Z).
expr(Z) --> num(X), [+], expr(Y), {Z is X + Y}.
expr(Z) --> num(X), [-], expr(Y), {Z is X - Y}.
num(D) --> [D], {number(D)}.

expr_compute(L, V)
   :- expr(V, L, []).
```

- first, create space for a value to be passed
- second, make the actual calculations!

Limitations

- Although this grammar expresses an equivalent language, its DCG does not work in Prolog.

```
expr --> num.
expr --> expr, [+], expr.
expr --> expr, [-], expr.
num --> [D], {number(D)}.
```

- DCGs have to be ***right-recursive***.

Limitations

- Although this grammar expresses an equivalent language, its DCG does not work in Prolog.

```
expr --> num.
expr --> expr, [+], expr.
expr --> expr, [-], expr.
num --> [D], {number(D)}.
```

- DCGs have to be ***right-recursive***.

- Furthermore, because Prolog descent is *amnesic*, it may inefficiently repeat the same computations (cf. **chart parsing**).

Extending DCGs

Top-down recognizer

```
:- consult('dcg2rules.pl').                    % np --> det, n. becomes rule(np, [det,n])
:- dcg2rules('naturalgrammarexample.pl').      % assert rule(np, [det,n])

tdr(Proto, Words) :-                           % Proto = list of non-terminals or words
    match(Proto, Words, [], []).               % success if beginning of Proto = Words

tdr([X|Proto], Words) :-
    rule(X, RHS),                              % retrieving rule that matches X
    append(RHS, Proto, NewProto),              % replacing X by RHS (= right-hand side)
    nl, write(X),write(' --> '), write(RHS),
    match(NewProto, Words, NewProto1, NewWords),
    tdr(NewProto1, NewWords).                                % lateral recursive call

match([X|L1], [X|L2], R1, R2) :-
    !,
    write('\t****  recognized: '), write(X),
    match(L1, L2, R1, R2).

match(L1, L2, L1, L2).
```

start from structures and fill them with words..

Bottom-up parsing

```
bup([s]).               % success when one gets s as a list of words

bup(P):-
   append(Pref, Rest, P),          % P is split into three pieces
   append(RHS, Suff, Rest),        % P = Pref + RHS + Suff
   rule(X, RHS),                   % bottom up use of rule
   append(Pref, [X|Suff], NEWP),   % RHS is replaced by X in P:
   bup(NEWP).                      % lateral recursive call
```

start from words
to fill structures...

Exploiting unification

- Using arguments we can perform additional checks, e.g. checking *number agreement*:

```
np(Number)  --> det(Number), n(Number).

det(singular) --> [a].
det(plural) --> [many].
det(_) --> [the].

n(singular)   --> [dog].
n(plural)   --> [dogs].
```

Exploiting unification

- Using arguments we can perform additional checks, e.g. checking *number agreement*:

```
np(Number) --> det(Number), n(Number).

det(singular) --> [a].
det(plural) --> [many].
det(_) --> [the].

n(singular) --> [dog].
n(plural) --> [dogs].
```

- but also *gender agreement*, *transitivity*, etc.

Exploiting unification

- Using arguments we can perform additional checks, e.g. checking *number agreement*:

```
np(Number) --> det(Number), n(Number).

det(singular) --> [a].
det(plural) --> [many].
det(_) --> [the].

n(singular) --> [dog].
n(plural) --> [dogs].
```

- but also *gender agreement*, *transitivity*, etc.
- but also "**semantic**" agreements (edible objects for eating, etc.)

Feature structures

- Feature structures may be described with lists. But an important improvement consists in using maps:

```
np([number:singular, person:3,
    gender:feminine,
    sentience:true]) --> [mary].
v([subj:[number:singular, person:3,
   gender:_, sentience:true],
   event:false]) --> [thinks].
v([subj:[number:singular, person:3,
   gender:_, sentience:_],
   event:true]) --> [falls].
```

Variable-length feature structures

- To have the possibility of not defining all elements of features structures, we consider unterminated lists

```
?- A = [number:singular, person:3,
        sentience:true, gender:feminine | _],
   B = [number:singular, person:3 | _],
   A = B.
```

Variable-length feature structures

- To have the possibility of not defining all elements of features structures, we consider unterminated lists

```
?- A = [number:singular, person:3,
        sentience:true, gender:feminine | _],
   B = [number:singular, person:3 | _],
   A = B.
```

- but we need to neglect their order still…

Variable-length feature structures

- To have the possibility of not defining all elements of features structures, we consider unterminated lists

```
?- A = [number:singular, person:3,
        sentience:true, gender:feminine | _],
   B = [number:singular, person:3 | _],
   A = B.
```

- but here their order still matters… so use this:

```
unify(FS, FS) :- !.

unify([ Feature | R1 ], FS) :-
  select(Feature, FS, FS1),
  !,
  unify(R1,FS1).
```

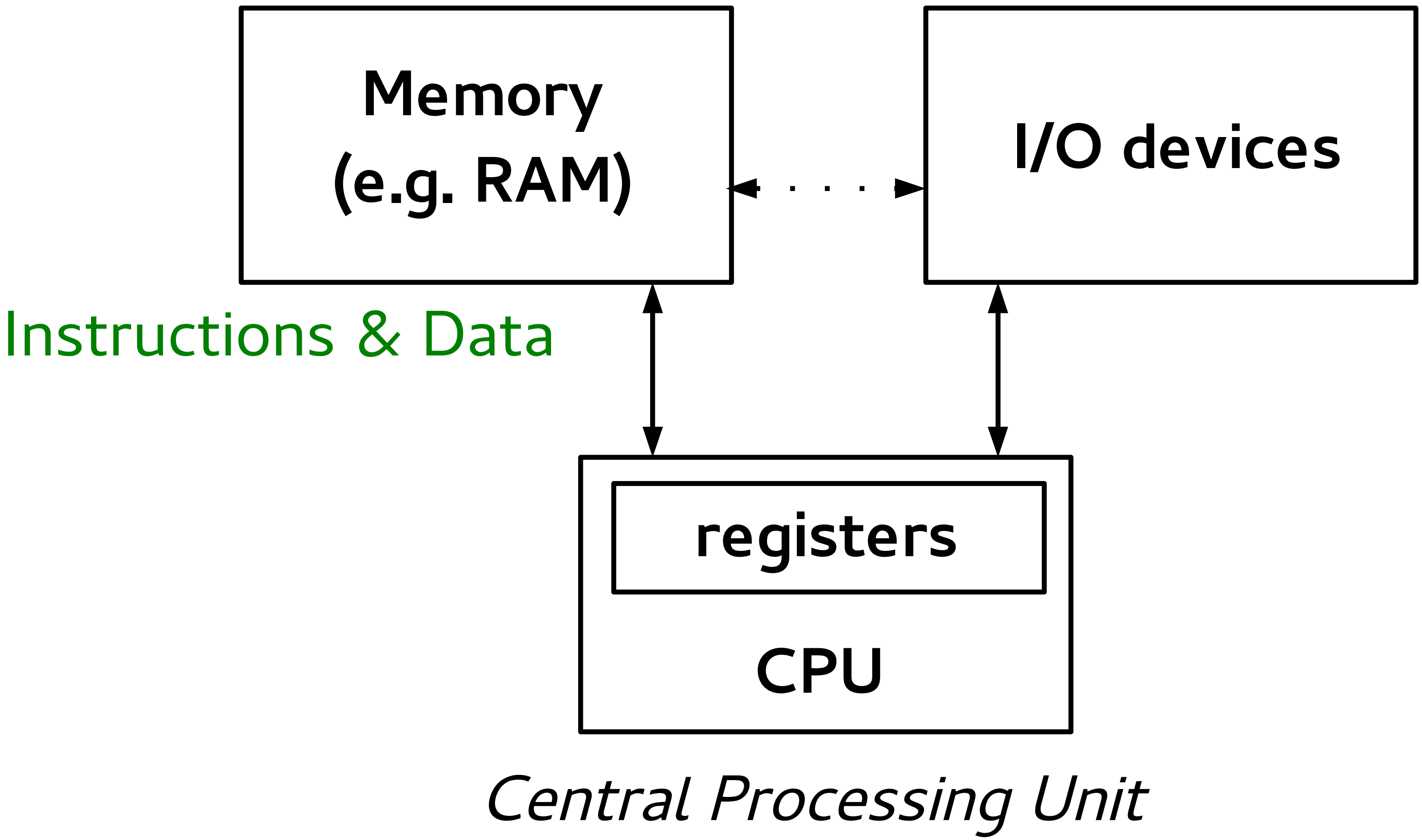
Meta-programming

Going *meta-*

- The prefix meta- is used to say the we go ”up” recursively on a concept:
  - **meta-physics**: physics of physics
  - **meta-data**: data about data
  - **meta-reasoning**: reasoning about reasoning
  - ...

A modern computer (roughly)

~ Von Neumann architecture



From a hardware perspective, **instructions *are* data!**

Meta-programming

- A meta-program is a program that manipulates other programs (or itself) as its data.

Meta-programming

- A meta-program is a program that manipulates other programs (or itself) as its data.
- Meta-programming is the act of writing meta-programs. Examples of meta-programs are:
  - when executed, print a copy of their codes
  - using an "eval" function to execute dynamically generated code
  - relying on *macros* (***generative programming***)
  - reasoning with their own structures and processes (e.g. reading the class of an object) (***reflection,*** namely ***introspection***)
  - compilers or interpreters of any language

Meta-programming

- A meta-program is a program that manipulates other programs (or itself) as its data.
- Why is it done?
  - to get around limitations of or to enhance with new features the primary development language,
  - to encapsulate domain-specific knowledge, by introducing a **domain-specific language** (DSL) with its own semantics
  - to allow users to configure a system in a easier way

Meta-programming in Prolog

- Let us start from the simplest meta-interpreter..

```
prove(Goal) :-
  call(Goal).
```

`call/1` is a built-in predicate invoking the parameter as goal

Meta-programming in Prolog

- Let us start from the simplest meta-interpreter..

```
prove(Goal) :-
  call(Goal).
```

object-level		meta-level
Goal	*going meta-* →	Goal
object		instruction
(term)		(prove me ...)

`call/1` is a built-in predicate invoking the parameter as goal

Meta-programming in Prolog

- Going further…

```
prove(true).

prove([Goal1, Goal2]) :-
  prove(Goal1),
  prove(Goal2).

prove(Goal) :-
  clause(Goal, Body),
  prove(Body).
```

`clause/2` is a built-in predicate true if Head can be unified with a clause head and Body with the corresponding clause body.

Meta-programming in Prolog

- Going further…

```
prove(true).

prove([Goal1, Goal2]) :-
  prove(Goal1),
  prove(Goal2).

prove(Goal) :-
  clause(Goal, Body),
  prove(Body).
```

Introspection

`clause/2` is a built-in predicate true if Head can be unified with a clause head and Body with the corresponding clause body.

Meta-programming in Prolog

- Using it for something more useful: *trace of proof*!

```
prove(true) :- !.

prove([Goal1, Goal2]) :- !,
  prove(Goal1),
  prove(Goal2).

prove(Goal) :-
  write('Call: '), write(Goal), nl,
  clause(Goal, Body),
  prove( Body),
  write('Exit: '), write( Goal), nl.
```

`clause/2` is a built-in predicate true if Goal can be unified with a clause head and Body with the corresponding clause body.