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](image1.jpg)

![Data (Star Trek)](image2.jpg)

![Samantha](image3.jpg)

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

---

ELIZA > (PLEASE TYPE IN ALL CAPS) WHAT’S YOUR NAME DEAR?
AMIT
ELIZA > HI, I’M ELIZA, WHAT DO YOU WANT TO TALK ABOUT?
AMIT > CAN I TALK ABOUT MY PROBLEM?
ELIZA > SURE...
AMIT > DUE TO ON GOING EXAMS IN COLLEGE, I HAVE A LOT OF STRESS.
ELIZA > PLEASE RELAX A LITTLE, SLEEP WELL.
AMIT > THANKS FOR YOUR ADVICE.
ELIZA > NO MENTION.
AMIT > BYE
ELIZA > BYE AND KEEP IN TOUCH...

---

ELIZA: Is something troubling you?
YOU: I am bored
ELIZA: Do you know anyone else who is bored?
YOU: Maybe
ELIZA: You aren’t sure?
YOU: I don’t really know
ELIZA: Why don’t you really know?
YOU: Well, I haven’t really asked
ELIZA: Do you say you haven’t really asked for some special reason?
YOU: I’m feeling blue
ELIZA: Do you believe it is normal to be feeling blue?
YOU: What’s normal?
ELIZA: What is it you really want to know?
YOU: Why don’t I have more energy in the morning?
ELIZA: That’s quite interesting.
YOU: Not really
ELIZA: Do you feel strongly about discussing such things?
YOU: Yes, I’m LONELY
ELIZA: Is it because you are lonely that you came to me?
YOU: Duh
ELIZA: Does talking about this bother you?
YOU: It is SO NICE to TALK TO YOU
ELIZA: We were discussing you -- not me.
YOU:
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)

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 θoʊ.../
(Phonology, the study of pronunciation)

go/go ing
(Morphology, the study of word constituents)

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

“|” =
(Semantics, the study of meaning)

It doesn't matter what I sing.
(Pragmatics, the study of language use)

• All these levels play a role with language!
not only in verbal language...

Vittore Carpaccio,
Due Dame, ~1495
Beware of context!

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

not only in verbal language...
Language Processing in Prolog
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 syntactic category
- \( \Sigma \) 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 \(( V \cup \Sigma )^* \) \((* = \text{Kleene star symbol})\)
- \( S \) 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
    - $\emptyset$ denoting the empty set: {} 
    - $\epsilon$ 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
    • \( \emptyset \) denoting the empty set: \{\}\n    • \( \varepsilon \) denoting the set containing only the empty string: \{""\}\n    • \( a \) 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 concatenating 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 \( \varepsilon \), 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: \( \{ 0^n1^n : n \in \mathbb{N} \} \) is not regular.
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:**

\[
\begin{align*}
\text{s} &\Rightarrow \text{np vp} & s(X, Z) &\leftarrow \text{np}(X, Y), \text{vp}(Y, Z). \\
\text{np} &\Rightarrow \text{det n} & \text{np}(X, Z) &\leftarrow \text{det}(X, Y), \text{n}(Y, Z). \\
\text{vp} &\Rightarrow \text{v np} & \text{vp}(X, Z) &\leftarrow \text{v}(X, Y), \text{np}(Y, Z). \\
\text{vp} &\Rightarrow \text{v} & \text{vp}(X, Z) &\leftarrow \text{v}(X, Z). \\
\text{det} &\Rightarrow \text{a} & \text{det}([\text{the}|W], W). \\
\text{det} &\Rightarrow \text{the} & \text{det}([\text{a}|W], W). \\
\text{n} &\Rightarrow \text{woman} & \text{n}([\text{woman}|W], W). \\
\text{n} &\Rightarrow \text{man} & \text{n}([\text{man}|W], W). \\
\text{v} &\Rightarrow \text{kisses} & \text{v}([\text{kisses}|W], W).
\end{align*}
\]

?- s([a,woman,kisses,a,man], []).
True.
From CFG to DCG

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

\[
\begin{align*}
\text{s} & \rightarrow \text{np(vp)} & \text{s} & \rightarrow \text{np, vp}.
\text{np} & \rightarrow \text{det(n)} & \text{np} & \rightarrow \text{det, n}.
\text{vp} & \rightarrow \text{v(np)} & \text{vp} & \rightarrow \text{v, np}.
\text{vp} & \rightarrow \text{v} & \text{vp} & \rightarrow \text{v}.
\text{det} & \rightarrow \text{a} & \text{det} & \rightarrow \text{[the]}.
\text{det} & \rightarrow \text{the} & \text{det} & \rightarrow \text{[a]}.
\text{n} & \rightarrow \text{woman} & \text{n} & \rightarrow \text{[woman]}.
\text{n} & \rightarrow \text{man} & \text{n} & \rightarrow \text{[man]}.
\text{v} & \rightarrow \text{kisses} & \text{v} & \rightarrow \text{[kisses]}.
\end{align*}
\]

?­ s([a,woman,kisses,a,man], []). True.
A computational example

Backus–Naur Form (BNF):

\[ \langle \text{expr} \rangle ::= \langle \text{num} \rangle \mid \langle \text{num} \rangle + \langle \text{expr} \rangle \mid \langle \text{num} \rangle - \langle \text{expr} \rangle \]

\[
\begin{align*}
\text{expr} & \Rightarrow \text{num} \\
\text{expr} & \Rightarrow \text{num} + \text{expr} \\
\text{expr} & \Rightarrow \text{num} - \text{expr} \\
\text{num} & \Rightarrow 0 \\
\text{num} & \Rightarrow 1 \\
\text{num} & \Rightarrow 2 \\
\text{num} & \Rightarrow \ldots
\end{align*}
\]
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.
um --> [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.

\[
\begin{align*}
\text{expr} & \rightarrow \text{num}. \\
\text{expr} & \rightarrow \text{expr}, [+], \text{expr}. \\
\text{expr} & \rightarrow \text{expr}, [-], \text{expr}. \\
\text{num} & \rightarrow [D], \{ \text{number}(D) \}.
\end{align*}
\]

• DCGs have to be \textit{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').
:- dcg2rules('naturalgrammarexample.pl').

% np --> det, n. becomes rule(np, [det,n])
% assert rule(np, [det,n])

tdr(Proto, Words) :-
    match(Proto, Words, [], []).

% Proto = list of non-terminals or words
% success if beginning of Proto = Words

tdr([X|Proto], Words) :-
    rule(X, RHS),
    append(RHS, Proto, NewProto),
    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

\[
bup([s]). \quad \text{% success when one gets } s \text{ as a list of words}
\]

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

\text{start from words to fill structures...}
Exploiting unification

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

  \[
  \text{np(}\text{Number}\text{)} \rightarrow \text{det(Number)}, \text{ n(}\text{Number}\text{)}. \\
  \text{det(}\text{singular}\text{)} \rightarrow [a]. \\
  \text{det(}\text{plural}\text{)} \rightarrow [many]. \\
  \text{det(}_\text{)\text{)} \rightarrow [the]. \\
  \text{n(}\text{singular}\text{)} \rightarrow [\text{dog}]. \\
  \text{n(}\text{plural}\text{)} \rightarrow [\text{dogs}].
  \]
Exploiting unification

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

  \[
  \text{np(}\text{Number}\text{)} \rightarrow \text{det(}\text{Number}\text{)}, \text{n(}\text{Number}\text{)}.
  \]

  \[
  \text{det(}\text{singular}\text{)} \rightarrow [a].
  \]
  \[
  \text{det(}\text{plural}\text{)} \rightarrow [\text{many}].
  \]
  \[
  \text{det(}_\text{}\text{)} \rightarrow [\text{the}].
  \]

  \[
  \text{n(}\text{singular}\text{)} \rightarrow [\text{dog}].
  \]
  \[
  \text{n(}\text{plural}\text{)} \rightarrow [\text{dogs}].
  \]

• but also *gender agreement, transitivity*, etc.
Exploiting unification

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

\[ np(\text{Number}) \rightarrow \text{det}(\text{Number}), \text{n}(\text{Number}). \]

\[ \text{det}(\text{singular}) \rightarrow [a]. \]
\[ \text{det}(\text{plural}) \rightarrow [\text{many}]. \]
\[ \text{det}(\_\_\_) \rightarrow [\text{the}]. \]

\[ \text{n}(\text{singular}) \rightarrow [\text{dog}]. \]
\[ \text{n}(\text{plural}) \rightarrow [\text{dogs}]. \]

• 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:

\[
\text{np}([\text{number:singular, person:3, gender:feminine, sentience:true}]) \rightarrow [\text{mary}].
\]

\[
\text{v}([\text{subj}:[\text{number:singular, person:3, gender:_, sentience:true}], event:false]) \rightarrow [\text{thinks}].
\]

\[
\text{v}([\text{subj}:[\text{number:singular, person:3, gender:_, sentience:_}], event:true]) \rightarrow [\text{falls}].
\]
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

```prolog
?- 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 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 an easier way
Meta-programming in Prolog

• Let us start from the simplest meta-interpreter.

\[
\text{prove}(\text{Goal}) :\text{-}
\text{call}(\text{Goal}).
\]

call/1 is a built-in predicate invoking the parameter as goal
Let us start from the simplest meta-interpreter..

\[
\text{prove}(\text{Goal}) :\text{-} \\
\text{call}(\text{Goal}).
\]

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).

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.