The Glue Semantics Workbench

Mark-Matthias Zymla
Moritz Messmer
University of Konstanz

LFG Conference, Vienna
17/07/2018

About

- **GOAL:** Providing a modular, easy to use glue semantics tool written in Java that is useful for both computational linguists and formal semanticists
  - Provide a tractable, efficient implementation of (a fragment of) linear logic
  - Modular system that can be connected to various NLP pipelines, in particular XLE (Crouch et al., 2017), Stanford CoreNLP (Manning et al., 2014) with minimal effort
  - Illustration of the system by means of a classic formal semantic phenomenon
  - Free to use open-source software

Some existing resources:

- NLTK computational semantics package (Python)
- glue implementation PARC by Richard Crouch and colleagues (Prolog)
- "Instant Glue" prover by Miltiadis Kokkonidis (Prolog)
- glue prover algorithm outlined by (Lev, 2007)

→ served as initial guiding points

Why Java?

- object-oriented paradigm fits resource-sensitive nature of linear logic
- possibility to modularize the program
- many interfaces to libraries like the Stanford CoreNLP tools
- Java virtual machines ubiquitous across all operating systems
- Java is widely used both in academic and industrial software development

Background on linear logic

"[glue semantics] is an approach to the semantic interpretation of natural language that uses a fragment of linear logic as a deductive glue for combining together the meanings of words and phrases"
–Crouch and van Genabith, (2000)

- linear logic (LL) is a *resource-conscious* logic
  premises, assumptions and conclusions as used in logical proofs are resources (not truths or facts)

$$A, A \rightarrow B, A \rightarrow C \models A, B, C \qquad \text{traditional}$$

$$\text{vs. } A, A \multimap B, A \multimap C \not\models A, B, C \qquad \text{LL}$$

- a sentence denotes a successful linear logic proof

$\rightarrow$ all resources introduced by the sentence have to be consumed

The appeal of linear logic

- syntax of proof systems of "traditional" logics is not always in one-to-one correspondence to the underlying proof object

$\rightarrow$ LL better suited to describe underlying proof objects

- resource usage occurs in natural language: Words and phrases correspond to resources
- (Certain fragments) can be implemented in a tractable manner

Some technicalities

- lexical entries consist of two elements:
  - **glue language:** linear logic – can be understood as semantic types (Curry-Howard-isomorphism)
  - **meaning language** Montague style semantics (but other formalism are possible)

ex. $\lambda x.sleep(x) : A \multimap B$

ex. $\lambda P.\lambda Q.\exists x[P(x) \wedge Q(x)] : (A \multimap B) \multimap ((C \multimap D) \multimap D)$

Relevant rules

- we use the *implicational fragment* of linear logic

Introduction rule

$$\frac{\begin{array}{c}[x : A]^i \\ \vdots \\ f(x) : B\end{array}}{\lambda x.f(x) : A \multimap B} \multimap_{I,i}$$

Elimination rule

$$\frac{f : A \multimap B \qquad a : A}{f(a) : B} \multimap_E$$

Semantic composition as proof

- *John loves Mary.*
- lexical entries:
  - $[\![\text{John}]\!] = j : g$
  - $[\![\text{Mary}]\!] = m : h$
  - $[\![\text{loves}]\!] = \lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f)$

$$\dfrac{\dfrac{\lambda x.\lambda y.loves(x, y) : g \multimap (h \multimap f) \qquad j : g}{\lambda y.loves(j, y) : h \multimap f} \qquad m : h}{loves(j, m) : f}$$

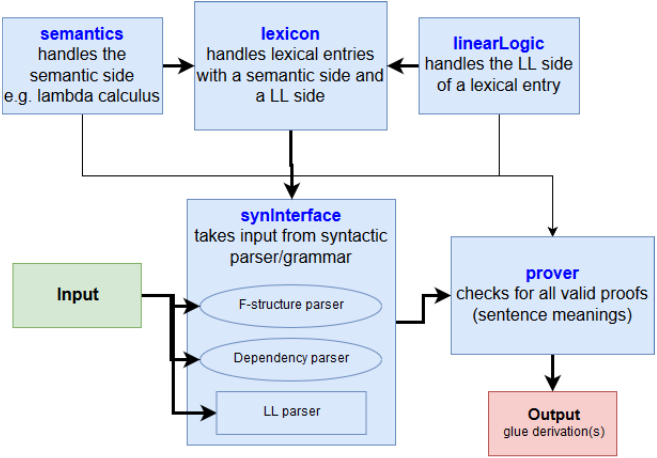
From syntax to semantics

$$\begin{bmatrix} \text{PRED} & \text{'love}\langle\text{John,Mary}\rangle\text{'} \\ \text{SUBJ} & \begin{bmatrix}\text{PRED} & \text{'John'}\end{bmatrix} \\ \text{OBJ} & \begin{bmatrix}\text{PRED} & \text{'Mary'}\end{bmatrix} \end{bmatrix}$$

- $\lambda x.\lambda y.loves(x, y)$: $\uparrow .SUBJ \multimap (\uparrow .OBJ \multimap \uparrow)$
- $j :\uparrow .SUBJ$
- $m :\uparrow .OBJ$

- $\uparrow$ refers to a specific f-structure node (e.g. $\uparrow$ points to the outer f-structure; $\uparrow .SUBJ$ points to the f-structure node of the subject)
- syntactic analysis determines linear logic resources (see e.g. Dalrymple, 2001 and subsequent work)
- traditionally co-descriptive, but description-by-analysis also possible (Kaplan, 1995)

Modules


semantics
handles the semantic side e.g. lambda calculus
lexicon
handles lexical entries with a semantic side and a LL side
linearLogic
handles the LL side of a lexical entry
synInterface
takes input from syntactic parser/grammar
F-structure parser
Dependency parser
LL parser
Input
prover
checks for all valid proofs (sentence meanings)
Output
glue derivation(s)

Prover algorithm

based on three principles, taken from algorithms by Hepple, (1996) and Gupta and Lamping, (1998)

I indexation

II compilation

III skeleton-modifier distinction

Hepple, (1996): Basic chart prover

Indexation

- Hepple parser stores partial results and re-uses them to prevent backtracking
  - linear use of resources enforced by using indexes
  - each LL formula (=**premise**) assigned unique index
  - when combining premises their index sets are unified
  - two premises can only be combined when their index sets are disjoint
- **Example:**

$j : g$	[0]
$\lambda x.sleeps(x) : g \multimap f$	[1]
$sleeps(j) : f$	[0,1]

first-order chart prover pseudo code

Stack A (agenda)
List D (database)
for A contains premises **do**
  pop premise P_A
  add P_A to D
  for all Premises P_D in D **do**
    if P_A and P_D combinable and index sets disjoint **then**
      add new combined premise to A
    end if
  end for
end for
if any P_D from D has a full set of indexes it is a valid solution

higher-order chart prover

Compilation

- higher-order formulas with nested consumers usually require $\multimap$-introduction
- hypothetical reasoning makes computation very complex
- Hepple's solution: transform the initial (potentially higher-order) formulas into a set of first-order formulas
- nested consumers are "compiled out" to additional assumptions:
  $(a \multimap b) \multimap c$ [0] $\Rightarrow$

higher-order chart prover

Compilation

- higher-order formulas with nested consumers usually require $\multimap$-introduction
- hypothetical reasoning makes computation very complex
- Hepple's solution: transform the initial (potentially higher-order) formulas into a set of first-order formulas
- nested consumers are "compiled out" to additional assumptions:

  $$(a \multimap b) \multimap c \ [0] \Rightarrow b[a] \multimap c \ [0]$$
  $$\{a\} \ [1]$$

Higher-order chart prover

- extracted assumptions are marked as such (notated with $\{\}$) and assigned a new unique index
- formula from which assumption is extracted gets extracted resource as discharge (notated with [ ])
- rules to assure that only the right premises combine:

Higher-order chart prover

- extracted assumptions are marked as such (notated with {}) and assigned a new unique index
- formula from which assumption is extracted gets extracted resource as discharge (notated with [ ])
- rules to assure that only the right premises combine:
  - if one or both premises contain assumptions, these are added to the set of assumptions of the combined premise
  - if a premise contains discharges, the set of assumptions of the other premise must contain the discharged resource
  - matched assumption and discharge pairs are removed from the book-keeping

Higher-order chart prover

- extracted assumptions are marked as such (notated with {}) and assigned a new unique index
- formula from which assumption is extracted gets extracted resource as discharge (notated with [ ])
- rules to assure that only the right premises combine:
    - if one or both premises contain assumptions, these are added to the set of assumptions of the combined premise
    - if a premise contains discharges, the set of assumptions of the other premise must contain the discharged resource
    - matched assumption and discharge pairs are removed from the book-keeping
- meaning side: compilation step amounts to functional application with deliberate "accidental binding" of relevant variable

Compilation and combination of higher-order formula

Deliberate accidental binding is a technical workaround to introducing and replacing temporary variables.

(1) Everybody sleeps.

original premises:

$g_1 \multimap f$: $\lambda y.\text{sleep}(y)$

$(g_2 \multimap H) \multimap H$: $\lambda P.\forall x[\text{person}(x) \land P(x)]$

Compilation and combination of higher-order formula

Deliberate accidental binding is a technical workaround to introducing and replacing temporary variables.

(1) Everybody sleeps.

original premises:

$g_1 \multimap f \quad : \lambda y.\text{sleep}(y)$

$(g_2 \multimap H) \multimap H \quad : \lambda P.\forall x[\text{person}(x) \wedge P(x)]$

compiled premises:

$g_1 \multimap f \quad : \lambda y.\text{sleep}(y)$

$\{g_2\} \quad : v$

$H[g_2] \multimap H \quad : \lambda u.\lambda P.\forall x[\text{person}(x) \wedge P(x)](\lambda v.u)$

Compilation and combination of higher-order formula

Deliberate accidental binding is a technical workaround to introducing and replacing temporary variables.

(1) Everybody sleeps.

original premises:

$$g_1 \multimap f \quad : \quad \lambda y.\text{sleep}(y)$$
$$(g_2 \multimap H) \multimap H \quad : \quad \lambda P.\forall x[\text{person}(x) \wedge P(x)]$$

compiled premises:

$$g_1 \multimap f \quad : \quad \lambda y.\text{sleep}(y)$$
$$\{g_2\} \quad : \quad v$$
$$H[g_2] \multimap H \quad : \quad \lambda u.\lambda P.\forall x[\text{person}(x) \wedge P(x)](\lambda v.u)$$

$$\dfrac{\dfrac{H[g_2] \multimap H : \lambda u.\lambda P.\forall x[\text{person}(x) \wedge P(x)](\lambda v.u) \qquad \dfrac{g_1 \multimap f : \lambda y.\text{sleep}(y) \quad \{g_2\} : v}{f\{g_2\} : \text{sleep}(v)}}{f : \lambda P.\forall x[\text{person}(x) \wedge P(x)](\lambda v.\text{sleep}(v))}[H/f]}{f : \forall x[\text{person}(x) \wedge \text{sleep}(x)]}\beta\text{-conversion}$$

Pseudo code: higher-order prover

```
Stack A (agenda)
List D (database)
Solutions S (all premises with full index sets)
for A contains premises do
    pop premise P_A
    add P_A to D
    for all Premises P_D in D do
        if P_A and P_D combinable and index sets disjoint then
            if P_A and/or P_D contain assumptions then
                combine sets of assumptions
                add new combined premise to A
            else if P_A or P_D contain discharges then
                if discharges are a subset of assumptions then
                    delete "used" discharges and assumptions
                    add new combined premise to A
                end if
            else
                no assumptions or discharges; combine premises as usual
            end if
        end if
    end for
end for
```

Treating modifiers (following Gupta and Lamping, 1998)

skeleton-modifier distinction

- adjuncts like adjectives, adverbs, etc. significantly increase the complexity of a deduction
- need to be treated separately to prevent explosion of partial results
- separation between two types of glue premises:
  - modifier: each positive (producer) occurrence of a resource paired with negative (consumer) occurrence
    $(v_{+} \multimap r_{-})_{-} \multimap (v_{-} \multimap r_{+})_{+} : \lambda P.\lambda x.P(x) \wedge black(x)$
  - skeleton: premise with "unmatched" producer/consumer resources
    $(v_{-} \multimap r_{+}) : \lambda x.dog(x)$
- modifiers do not need to be compiled
- for each new skeleton premise taken from the agenda, check potential combination with modifiers

Syntax/semantics correspondence: quantifiers

Determiners

- the template for quantifiers is: $(x \multimap RESTR) \multimap ((SCOPE \multimap\uparrow) \multimap\uparrow)$.
- the restrictor is always the dependency that governs the quantifier
- the scope is newly instantiated for a quantifier and later unified with the arguments of the verb.
  - g: $(x \multimap SUBJ) \multimap ((SCOPE_A \multimap\uparrow) \multimap\uparrow)$
  - h: $(x \multimap OBJ) \multimap ((SCOPE_B \multimap\uparrow) \multimap\uparrow)$
  - $g \multimap (h \multimap f)$: $SCOPE_A \multimap (SCOPE_B \multimap\uparrow)$

Deriving ambiguities with the glue prover

An example with a quantifier scope ambiguity:

(2) A dog chases every cat.

dog	$(g \multimap g) : \lambda y_e.dog(y)$	[0]
a	$((g \multimap g) \multimap ((h \multimap Y) \multimap Y)) : \lambda P_{<e,t>}.\lambda Q_{<e,t>}.\exists x[P(x) \wedge Q(x)]$	[1]
cat	$(i \multimap i) : \lambda x'_e.cat(x')$	[2]
every	$((i \multimap i) \multimap ((j \multimap X') \multimap X')) : \lambda R_{<e,t>}.\lambda S_{<e,t>}.\forall z[R(z) \rightarrow S(z)]$	[3]
chase	$(h \multimap (j \multimap f)) : \lambda y'_e.\lambda z'_e.chases(y', z')$	[4]

Deriving ambiguities with the glue prover

After compilation we have the following premises:

(3) A dog chases every cat

$(g \multimap g) : \lambda y_e.dog(y)$ [0]

$(Y[h] \multimap (g[g] \multimap Y))$:
$\lambda t_t.\lambda s_t.\lambda Q_{<e,t>}.\lambda P_{<e,t>}.\exists x[P(x) \wedge Q(x)](\lambda x''_e.s)(\lambda y''_e.t)$ [1]

$\{g\} : x''$ [5]

$\{h\} : y''$ [6]

$(i \multimap i) : \lambda x'_e.cat(x')$ [2]

$(X'[j] \multimap (i[i] \multimap X'))$:
$\lambda v_t.\lambda u_t.\lambda S_{<e,t>}.\lambda R_{<e,t>}.\forall z[R(z) \rightarrow S(z)](\lambda z''_e.u)(\lambda x'''_e.v)$ [3]

$\{i\} : z''$ [7]

$\{j\} : x'''$ [8]

$(h \multimap (j \multimap f)) : \lambda y'_e.\lambda z'_e.chases(y', z')$ [4]

Deriving ambiguities with the glue prover

$$\dfrac{\dfrac{(h \multimap (j \multimap f)) : \lambda y'_e.\lambda z'_e.chases(y', z')[4] \qquad \{h\} : y''[6]}{(j \multimap f)\{h\} : \lambda z'_e.chases(y'', z')[4, 6]} \qquad \{j\} : x'''[8]}{f\{j, h\} : chases(y'', x''')[4, 6, 8]}$$

Deriving ambiguities with the glue prover

$$\dfrac{\dfrac{(h \multimap (j \multimap f)) : \lambda y'_e.\lambda z'_e.chases(y', z')[4] \qquad \{h\} : y''[6]}{(j \multimap f)\{h\} : \lambda z'_e.chases(y'', z')[4, 6]} \qquad \{j\} : x'''[8]}{f\{j, h\} : chases(y'', x''')[4, 6, 8]}$$

surface scope reading:

$$\dfrac{\dfrac{\dfrac{i\{i\}[2,7] \qquad \dfrac{f\{j,h\}[4,6,8] \qquad (X'[j] \multimap (i[i] \multimap X'))[3]}{(i[i] \multimap f)\{h\}[3,4,6,8]}}{f\{h\}[2,3,4,6,7,8]} \qquad (Y[h] \multimap (g[g] \multimap Y))[1]}{(g[g] \multimap f)[1,2,3,4,6,7,8]} \qquad g\{g\}[0,5]}{f : \exists x[dog(x) \wedge \forall z[cat(z) \rightarrow chases(x,z)]][0,1,2,3,4,5,6,7,8]}$$

Deriving ambiguities with the glue prover

$$\dfrac{\dfrac{(h \multimap (j \multimap f)) : \lambda y'_e.\lambda z'_e.chases(y',z')[4] \qquad \{h\} : y''[6]}{(j \multimap f)\{h\} : \lambda z'_e.chases(y'',z')[4,6]} \qquad \{j\} : x'''[8]}{f\{j,h\} : chases(y'',x''')[4,6,8]}$$

surface scope reading:

$$\dfrac{\dfrac{\dfrac{i\{i\}[2,7] \qquad \dfrac{f\{j,h\}[4,6,8] \qquad (X'[j] \multimap (i[i] \multimap X'))[3]}{(i[i] \multimap f)\{h\}[3,4,6,8]}}{f\{h\}[2,3,4,6,7,8]} \qquad (Y[h] \multimap (g[g] \multimap Y))[1]}{(g[g] \multimap f)[1,2,3,4,6,7,8]} \qquad g\{g\}[0,5]}{f : \exists x[dog(x) \wedge \forall z[cat(z) \rightarrow chases(x,z)]][0,1,2,3,4,5,6,7,8]}$$

inverse scope reading:

$$\dfrac{\dfrac{\dfrac{g\{g\}[0,5] \qquad \dfrac{f\{j,h\}[4,6,8] \qquad (Y[h] \multimap (g[g] \multimap Y))[1]}{(g[g] \multimap f)\{j\}[1,4,6,8]}}{f\{j\}[0,1,4,5,6,8]} \qquad (X'[j] \multimap (i[i] \multimap X'))[3]}{(i[i] \multimap f)[0,1,3,4,5,6,8]} \qquad i\{i\}[2,7]}{f : \forall z[cat(z) \rightarrow \exists x[dog(x) \wedge chases(x,z)]][0,1,2,3,4,5,6,7,8]}$$

The Glue Semantics Workbench in action

```
Selected file chase_webXLE.pl
[(g ⊸ g) : λx_e.cat(x)[0], ((g ⊸ g) ⊸ ((h ⊸ Y) ⊸ Y)) : λP_<e,t>.λQ_<e,t>.∀y[P(y) → Q(y)][1], (i ⊸ i) : λz_e.dog(z)[2],
Searching for valid proofs...
Agenda: [(g ⊸ g) : λx_e.cat(x)[0], {g} : x''[5], {h} : y''[6], (Y[h] ⊸ (g[g] ⊸ Y)) : λt_t.λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y)
Combining premises {g} : x''[5] and (g ⊸ g) : λx_e.cat(x)[0]
--> g{g} : cat(x'')[0, 5]
Combining premises {i} : z''[7] and (i ⊸ i) : λz_e.dog(z)[2]
--> i{i} : dog(z'')[2, 7]
Combining premises (j ⊸ (h ⊸ f)) : λy'_e.λz'_e.chase(y',z')[4] and {j} : x'''[8]
-->(h ⊸ f){j} : λz'_e.chase(x''',z')[4, 8]
Combining premises (h ⊸ f){j} : λz'_e.chase(x''',z')[4, 8] and {h} : y''[6]
-->f{j,h} : chase(x''',y'')[4, 6, 8]
Combining premises f{j,h} : chase(x''',y'')[4, 6, 8] and (X'[j] ⊸ (i[i] ⊸ X')) : λv_t.λu_t.λS_<e,t>.λR_<e,t>.∃x'[R(x') ∧
--> (i[i] ⊸ f){h} : λu_t.λS_<e,t>.λR_<e,t>.∃x'[R(x') ∧ S(x')](λz''_e.u)(λx'''_e.chase(x''',y''))[3, 4, 6, 8]
Combining premises f{j,h} : chase(x''',y'')[4, 6, 8] and (Y[h] ⊸ (g[g] ⊸ Y)) : λt_t.λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y) → Q(
--> (g[g] ⊸ f){j} : λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y) → Q(y)](λx''_e.s)(λy''_e.chase(x''',y''))[1, 4, 6, 8]
Combining premises (g[g] ⊸ f){j} : λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y) → Q(y)](λx''_e.s)(λy''_e.chase(x''',y''))[1, 4, 6, 8]
-->f{j} : ∀y[cat(y) → chase(x''',y)][0, 1, 4, 5, 6, 8]
Combining premises f{j} : ∀y[cat(y) → chase(x''',y)][0, 1, 4, 5, 6, 8] and (X'[j] ⊸ (i[i] ⊸ X')) : λv_t.λu_t.λS_<e,t>.λI
--> (i[i] ⊸ f) : λu_t.λS_<e,t>.λR_<e,t>.∃x'[R(x') ∧ S(x')](λz''_e.u)(λx'''_e.∀y[cat(y) → chase(x''',y)])[0, 1, 3, 4, 5,
Combining premises (i[i] ⊸ f) : λu_t.λS_<e,t>.λR_<e,t>.∃x'[R(x') ∧ S(x')](λz''_e.u)(λx'''_e.∀y[cat(y) → chase(x''',y)])
-->f : ∃x'[dog(x') ∧ ∀y[cat(y) → chase(x',y)]][0, 1, 2, 3, 4, 5, 6, 7, 8]
Combining premises (i[i] ⊸ f){h} : λu_t.λS_<e,t>.λR_<e,t>.∃x'[R(x') ∧ S(x')](λz''_e.u)(λx'''_e.chase(x''',y''))[3, 4, 6,
-->f{h} : ∃x'[dog(x') ∧ chase(x',y'')][2, 3, 4, 6, 7, 8]
Combining premises f{h} : ∃x'[dog(x') ∧ chase(x',y'')][2, 3, 4, 6, 7, 8] and (Y[h] ⊸ (g[g] ⊸ Y)) : λt_t.λs_t.λQ_<e,t>.λP
--> (g[g] ⊸ f) : λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y) → Q(y)](λx''_e.s)(λy''_e.∃x'[dog(x') ∧ chase(x',y'')])[1, 2, 3, 4, 6, 7
Combining premises (g[g] ⊸ f) : λs_t.λQ_<e,t>.λP_<e,t>.∀y[P(y) → Q(y)](λx''_e.s)(λy''_e.∃x'[dog(x') ∧ chase(x',y'')])[1
-->f : ∀y[cat(y) → ∃x'[dog(x') ∧ chase(x',y)]][0, 1, 2, 3, 4, 5, 6, 7, 8]
Found valid deduction(s):
f : ∃x'[dog(x') ∧ ∀y[cat(y) → chase(x',y)]][0, 1, 2, 3, 4, 5, 6, 7, 8]
f : ∀y[cat(y) → ∃x'[dog(x') ∧ chase(x',y)]][0, 1, 2, 3, 4, 5, 6, 7, 8]
Done!
```

From f-structures to glue premises

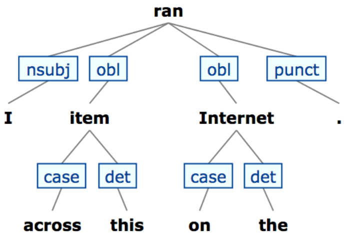
```
cf(1,eq(attr(var(0),'PRED'),semform('eat',2,[var(10),var(2)],[]))),
cf(1,eq(attr(var(0),'SUBJ'),var(10))),
cf(1,eq(attr(var(0),'OBJ'),var(2))),
...
cf(1,eq(attr(var(10),'PRED'),var(14))),
cf(1,eq(var(14),semform('Pluto',0,[],[]))),
...
cf(1,eq(attr(var(2),'PRED'),semform('bone',6,[],[]))),
```

- pattern-based parser extracts all grammatical functions and their PRED-values
- The system first generates lexical entries for grammatical functions and then generates the verbal spine

→ Description-by-analysis

→ May be outsourced to XLE transfer system

From dependencies to glue premises



- in LFG we make use of the flat f-structure to determine relations between syntax and semantics

→ We can simply flatten the dependency structure into a list of dependency facts using the underlying similarities of the two formalisms

Summary

- We presented a semantic parser at the core of which is a chart prover for linear logic formulas that decomposes higher order linear logic formulas into first order formulas.
- We implemented corresponding semantics that can be applied to natural language.
- We provide a small system for translating dependency parses and Prolog f-structure files into default semantic premises that can be proven/composed with the parser.
- The program can be easily extended/modified:
  - lexicon: implementing a proper, potentially co-descriptive lexicon
  - semantics: hook up with various semantic formalisms (e.g. DRT)

References I

Crouch, Dick et al. (2017). *XLE Documentation*. Palo Alto Research Center.

Crouch, Richard and Josef van Genabith (2000). *Linear Logic for Linguists: ESSLLI-2000 course notes. ESSLLI, Birmingham, UK.* Birmingham, UK.

Dalrymple, Mary (2001). *Lexical Functional Grammar*. Vol. 34. Academic Press. ISBN: 9780126135343.

Gupta, Vineet and John Lamping (1998). "Efficient linear logic meaning assembly". In: *Proceedings of the 17th international conference on Computational linguistics-Volume 1*. Association for Computational Linguistics, pp. 464–470.

Hepple, Mark (1996). "A compilation-chart method for linear categorial deduction". In: *Proceedings of the 16th conference on Computational linguistics-Volume 1*. Association for Computational Linguistics, pp. 537–542.

References II

Kaplan, Ronald M (1995). "The Formal Architecture of Lexical-Functional Grammar". In: *Formal Issues in Lexical-Functional Grammar* 47, pp. 7–27.

Lev, Iddo (2007). "Packed computation of exact meaning representations". PhD thesis. Stanford University.

Manning, Christopher et al. (2014). "The Stanford CoreNLP Natural Language Processing Toolkit". In: *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55–60.