Prototyping Natural Language Understanding with GF
Adding Semantics and Inference

Jan Frederik Schaefer

FAU Erlangen-Nürnberg

GF Summer School 2021
Singapore/online
August 2, 2021

whoami

- Just started my PhD (*Semantics Extraction in STEM*)
- FAU (**F**riedrich-**A**lexander-**U**niversität Erlangen-Nürnberg) in Germany
- KWARC research group:
  - Led by Michael Kohlhase
  - Knowledge representation and reasoning techniques
  - Focus on mathematical content

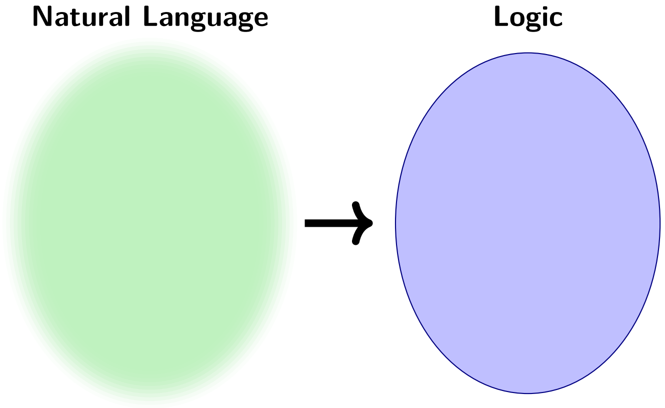
Talk in Stellenbosch:

- GF + MMT for semantics
- Mathematical language

Today:

- Extended and more mature system
- Less mathematical language

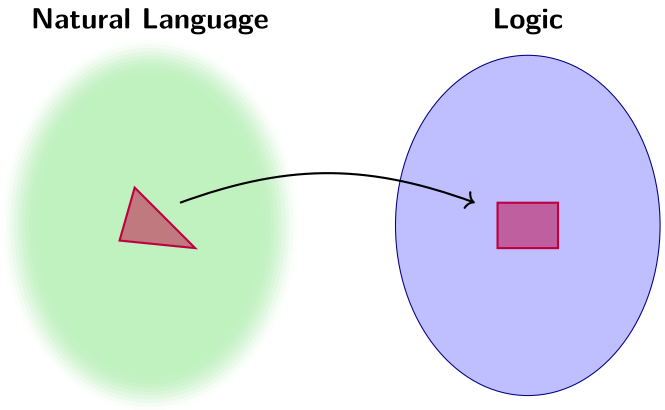
Method of Fragments



How do we get from messy language to formal logic?

Montague [Mon70]: Look at a “nice” subset and map into logic.

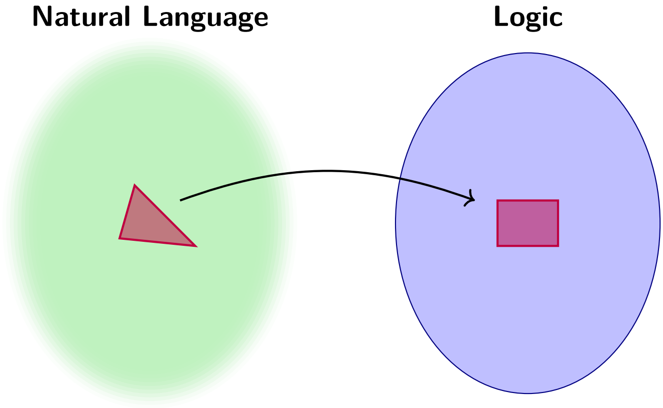
Method of Fragments



How do we get from messy language to formal logic?

Montague [Mon70]: Look at a “nice” subset and map into logic.

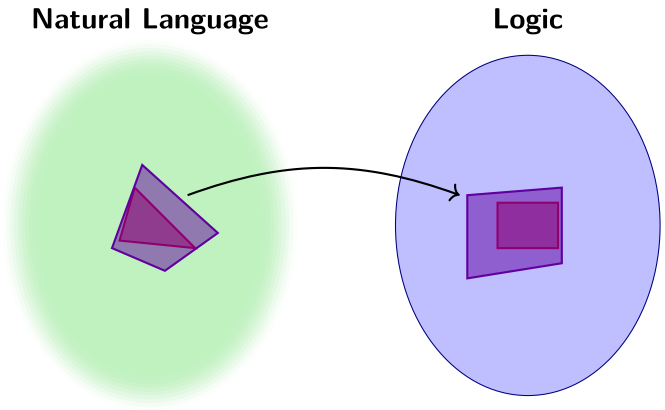
Method of Fragments



"Ahmed paints and Berta is quiet." $p(a) \wedge q(b)$

"Ahmed doesn't paint." $\neg p(a)$

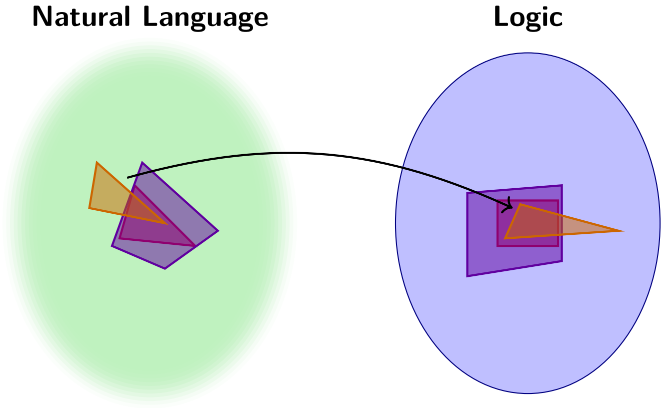
Method of Fragments



“Every student paints and is quiet.” $\forall x. s(x) \Rightarrow (p(x) \wedge q(x))$

“Nobody paints.” $\neg \exists x. p(x)$

Method of Fragments



"Ahmed isn't allowed to paint." $\neg\Diamond p(a)$

"Ahmed and Berta must paint." $(\Box p(a)) \wedge \Box p(b)$

Method of Fragments

Hand-waving is problematic:

"Ahmed paints. He is quiet." $\overset{?}{\rightsquigarrow}$ $p(a) \wedge q(a)$

Montague: Specify

- grammar, *fixes NL subset*
- target logic,
- semantics construction. *maps parse trees to logic*

Example from [Mon74]

T11. If $\phi, \psi \in P_t$ and ϕ, ψ translate into ϕ', ψ' respectively, then ϕ **and** ψ translates into $[\phi \wedge \psi]$, ϕ **or** ψ translates into $[\phi \vee \psi]$.

T12. If $\gamma, \delta \in P_{IV}$ and γ, δ translate into γ', δ' respectively, then γ **and** δ translates into $\hat{x}[\gamma'(x) \wedge \delta'(x)]$, γ **or** δ translates into $\hat{x}[\gamma'(x) \vee \delta'(x)]$.

T13. If $\alpha, \beta \in P_T$ and α, β translate into α', β' respectively, then α **or** β translates into $\hat{P}[\alpha'(P) \vee \beta'(P)]$.

Claim: That doesn't scale well $\rightsquigarrow$ **We need prototyping!**

Method of Fragments

How can we implement such fragments (with the help of GF)?

- Use GF for parsing
- 4 ideas how to continue

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{???}} \text{Logical Expr.}$$


```
parse -lang=Eng "Ahmed paints and Berta is quiet" | linearize -lang=Logic
p ( a ) ∧ q ( b )
```

Idea 1: Concrete Syntax for Logic

$$\mathrm{String}_{\mathrm{Eng}} \xrightarrow{\texttt{parse}} \mathrm{AST} \xrightarrow{\texttt{linearize}} \mathrm{String}_{\mathrm{Logic}}$$

```
parse -lang=Eng "Ahmed paints and Berta is quiet" | linearize -lang=Logic
p ( a ) ∧ q ( b )
```

```
abstract Grammar = {
  -- ...
  fun
    and_S : S -> S -> S;
    makeS : NP -> VP -> S;
    ahmed : NP;
    paint : VP;
   -- ...
}
```

```
concrete GrammarLogic of Grammar = {
  -- ...
  lin
    and_S s1 s2 = s1 ++ "∧" ++ s2;
    makeS n v = v ++ "(" ++ n ++ ")";
    ahmed = "a";
    paint = "p";
  -- ...
}
```

Idea 1: Concrete Syntax for Logic

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{linearize}} \text{String}_{\text{Logic}}$$

```
parse -lang=Eng "Ahmed is quiet and paints" | linearize -lang=Logic
q ( a ) ∧ p ( a ) ???
```

```
abstract Grammar = {
  -- ...
  fun
    and_S : S -> S -> S;
    makeS : NP -> VP -> S;
    ahmed : NP;
    paint : VP;
   -- ...
}
```

```
concrete GrammarLogic of Grammar = {
  -- ...
  lin
    and_S s1 s2 = s1 ++ "∧" ++ s2;
    makeS n v = v ++ "(" ++ n ++ ")";
    ahmed = "a";
    paint = "p";
  -- ...
}
```

Idea 1: Concrete Syntax for Logic

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{linearize}} \text{String}_{\text{Logic}}$$

```
parse -lang=Eng "Ahmed is quiet and paints" | linearize -lang=Logic
(λx. q (x) ∧ p (x)) ( a )
```

$\rightsquigarrow_\beta q(a) \wedge p(a)$

```
abstract Grammar = {
  -- ...
  fun
    and_S  : S -> S -> S;
    makeS  : NP -> VP -> S;
    ahmed  : NP;
    paint  : VP;
    and_VP : VP -> VP -> VP;
}
```

```
concrete GrammarLogic of Grammar = {
  -- ...
  lin
    and_S s1 s2 = s1 ++ "∧" ++ s2;
    makeS np vp = vp ++ "(" ++ np ++ ")";
    ahmed = "a";
    paint = "p";
    and_VP p q = "(λx."++p++"(x)"++
                       "∧"++q++"(x))"; }
```

Idea 2: Abstract Syntax for Logic

$$\mathrm{String}_{\mathrm{Eng}} \xrightarrow{\texttt{parse}} \mathrm{AST}_{\mathrm{NL}} \xrightarrow{\texttt{pt -compute}} \mathrm{AST}_{\mathrm{Logic}}$$

```
abstract Logic = {
  cat
    Prop; Term;
  fun
    and : Prop -> Prop -> Prop;
    a : Term;
    b : Term;
    p : Term -> Prop;
    q : Term -> Prop;
}
```

Idea 2: Abstract Syntax for Logic

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST}_{\text{NL}} \xrightarrow{\texttt{pt -compute}} \text{AST}_{\text{Logic}}$$

```
abstract Logic = {
  cat
    Prop; Term;
  fun
    and : Prop -> Prop -> Prop;
    a : Term;
    b : Term;
    p : Term -> Prop;
    q : Term -> Prop;
}
```

```
abstract Semantics = Grammar, Logic ** {
  fun
    t_S : S -> Prop;
    t_NP : NP -> Term;
    t_VP : VP -> (Term -> Prop);
  def
    t_NP ahmed = a;
    t_S (and_S s1 s2) = and (t_S s1) (t_S s2);
    t_VP (and_VP v1 v2) =
        \x -> and (t_VP v1 x) (t_VP v2 x);
```

Idea 2: Abstract Syntax for Logic

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST}_{\text{NL}} \xrightarrow{\texttt{pt -compute}} \text{AST}_{\text{Logic}}$$

```
abstract Logic = {
  cat
    Prop; Term;
  fun
    and : Prop -> Prop -> Prop;
    a : Term;
    b : Term;
    p : Term -> Prop;
    q : Term -> Prop;
}
```

```
abstract Semantics = Grammar, Logic ** {
  fun
    t_S : S -> Prop;
    t_NP : NP -> Term;
    t_VP : VP -> (Term -> Prop);
  def
    t_NP ahmed = a;
    t_S (and_S s1 s2) = and (t_S s1) (t_S s2);
    t_VP (and_VP v1 v2) =
        \x -> and (t_VP v1 x) (t_VP v2 x);
```

```
parse -lang=Eng -cat=Prop "Ahmed is quiet and paints" | put_tree -compute
and (q a) (p a)
```

All 4 Ideas

1. **Use another concrete syntax:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{linearize}} \text{String}_{\text{Logic}}$$

2. **Use another abstract syntax:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST}_{\text{NL}} \xrightarrow{\texttt{pt -compute}} \text{AST}_{\text{Logic}}$$

All 4 Ideas

1. **Use another concrete syntax:**

   $\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{linearize}} \text{String}_{\text{Logic}}$

2. **Use another abstract syntax:**

   $\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST}_{\text{NL}} \xrightarrow{\texttt{pt -compute}} \text{AST}_{\text{Logic}}$

3. **Use a general-purpose programming language:**

   $\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\text{Haskell code}} \text{Logical Expr.}$

All 4 Ideas

1. **Use another concrete syntax:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\texttt{linearize}} \text{String}_{\text{Logic}}$$

2. **Use another abstract syntax:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST}_{\text{NL}} \xrightarrow{\texttt{pt -compute}} \text{AST}_{\text{Logic}}$$

3. **Use a general-purpose programming language:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\text{Haskell code}} \text{Logical Expr.}$$

4. **Use a logic development framework:**

$$\text{String}_{\text{Eng}} \xrightarrow{\texttt{parse}} \text{AST} \xrightarrow{\text{MMT stuff}} \text{MMT term}$$

MMT

- Tool for logic development and knowledge representation
- Developed by KWARC group *we are biased towards MMT*
- "Bring your own logic"
- Implement syntax, semantics, calculi *"rapid prototyping"*
- Lots of logics already implemented *LATIN2 project*
- Supports type theory underlying GF
- Three steps:
  1. Represent abstract syntax in MMT *is automated*
  2. Declare target logic
  3. Declare mapping from 1. to 2.

MMT

1. **Represent abstract syntax in MMT** *is automated*
2. Declare target logic
3. Declare mapping from 1. to 2.

```
abstract Grammar = {
  cat
    S;
    VP;
    NP;
  fun
    makeS : NP -> VP -> S;
}

abstract More = Grammar ** {
  fun
    ahmed : NP;
    paint : VP;
}
```

```
theory Grammar =

    S  : type
    VP : type
    NP : type

    makeS : NP ⟶ VP ⟶ S


theory More =
    include Grammar
    ahmed : NP
    paint : VP
```

MMT

1. Represent abstract syntax in MMT *is automated*
2. **Declare target logic**
3. Declare mapping from 1. to 2.

```
theory Logic =
    o : type    // propositions
    ¬ : o → o
    ∧ : o → o → o
    ∨ : o → o → o
      = λa,b.¬ (¬ a ∧ ¬ b)

    ι : type    // terms
    a : ι
    p : ι → o
```

MMT

1. Represent abstract syntax in MMT *is automated*
2. **Declare target logic**
3. Declare mapping from 1. to 2.

```
theory Logic =
    o : type      // propositions
    ¬ : o → o
    ∧ : o → o → o
    ∨ : o → o → o
      = λa,b.¬ (¬ a ∧ ¬ b)

    ι : type      // terms
    a : ι
    p : ι → o
```

```
theory Logic : ur:?LF =
    o   : type ❙        // propositions ❙
    not : o → o         ❘ # ¬ 1 prec 100 ❙
    and : o → o → o ❘ # 1 ∧ 2 prec 80 ❙
    or  : o → o → o ❘ # 1 ∨ 2 prec 60 ❘
        = [a,b] ¬ (¬ a ∧ ¬ b) ❙

    ι : type ❙        // terms ❙
    a : ι ❙
    p : ι → o ❙
❚
```

MMT

1. Represent abstract syntax in MMT *is automated*
2. Declare target logic
3. **Declare mapping from 1. to 2.**

```
view Semantics : Grammar -> Logic =
    S = o
    NP = ι
    VP = ι ⟶ o
    and_S = λx,y.x∧y
    makeS = λn,v.v n
    ahmed = a
    paint = p
    // ...
```

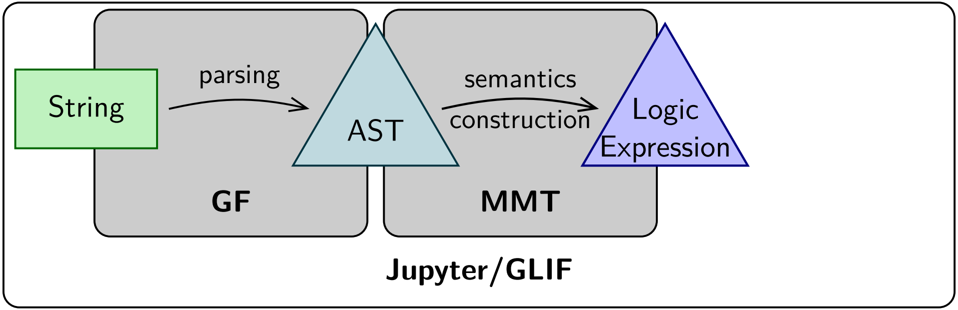
MMT

1. Represent abstract syntax in MMT *is automated*
2. Declare target logic
3. Declare mapping from 1. to 2.

```
view Semantics : Grammar -> Logic =
    S = o
    NP = ι
    VP = ι ⟶ o
    and_S = λx,y.x∧y
    makeS = λn,v.v n
    ahmed = a
    paint = p
    // ...
```

```
parse -lang=Eng "Ahmed is quiet and paints" | construct view=Semantics
q(a)∧p(a)
```

GLIF: The Grammatical Logical Inference Framework



GLIF: The Grammatical Logical Inference Framework

In [18]:

```
// the semantics construction maps symbols in the grammar to symbols in the logic. It

view Semantics : ?Grammar -> ?Logic =
    S = o ❙
    NP = ι ❙
    VP = ι → o ❙
    and_VP = [v1,v2] [x] (v1 x) ∧ (v2 x) ❙
    and_S = [s1,s2] s1 ∧ s2 ❙
    makeS = [n,v] v n ❙
    ahmed = a ❙
    berta = b ❙
    paint = p ❙
    be_quiet = q ❙
❚
```

Successfully imported Semantics.mmt

In [19]:

```
parse -lang=Eng "Ahmed is quiet and paints" | construct -no-simplify
```

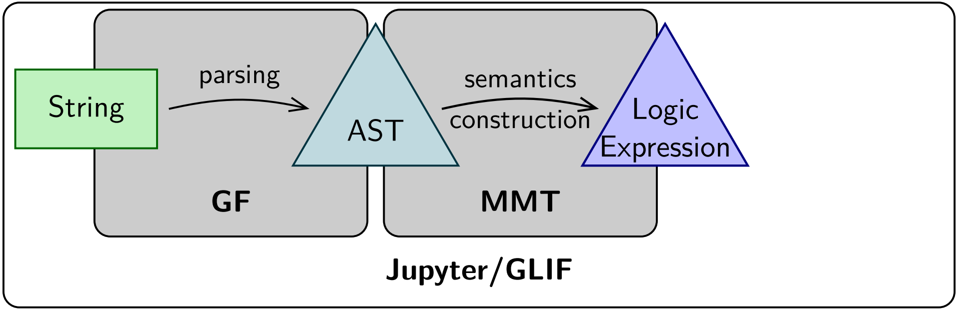
([n,v]v n) a (([v1,v2][x](v1 x) ∧ (v2 x)) q p)

In [20]:

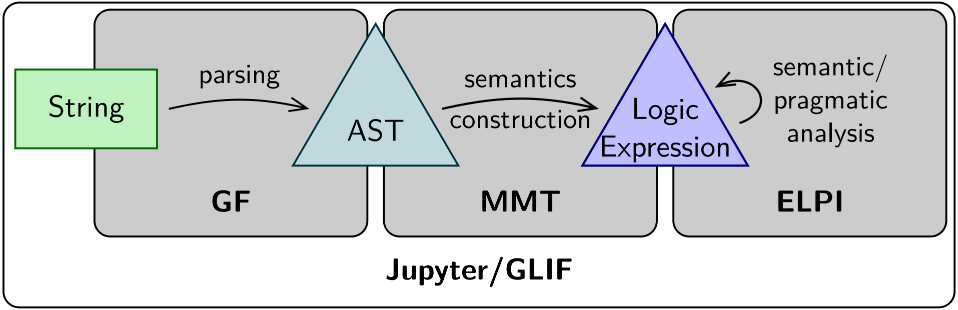
```
parse -lang=Eng "Ahmed is quiet and paints" | construct
```

(q a) ∧ (p a)

GLIF: The Grammatical Logical Inference Framework



GLIF: The Grammatical Logical Inference Framework



GF	(= **grammar** framework)
+ **MMT**	(= **logic** framework)
+ **ELPI**	(= **inference** framework)
= **GLIF**	(= **natural language understanding** framework)

ELPI

- Extension of λProlog *supports higher-order abstract syntax*
- Generic inference/reasoning step after semantics construction
- Goal: Use it for semantic/pragmatic analysis

ELPI

- Extension of λProlog *supports higher-order abstract syntax*
- Generic inference/reasoning step after semantics construction
- Goal: Use it for semantic/pragmatic analysis

Example: Discard wrong readings in controlled natural language

"the ball has a mass of 5kg" $\longrightarrow$ AST $\longrightarrow$ $\mathrm{mass}(\mathrm{theball}, \mathrm{quant}(5, \mathrm{kilo\ gram}))$

ELPI

- Extension of λProlog *supports higher-order abstract syntax*
- Generic inference/reasoning step after semantics construction
- Goal: Use it for semantic/pragmatic analysis

Example: Discard wrong readings in controlled natural language

"a kinetic energy of 12mN"

$\to \text{AST}_1 \longrightarrow \lambda x.E_{\mathsf{kin}}(x, \mathsf{quant}(2, \textbf{milli Newton}))$

$\to \text{AST}_2 \longrightarrow \lambda x.E_{\mathsf{kin}}(x, \mathsf{quant}(2, \textbf{meter}\cdot\textbf{Newton}))$

ELPI

- Extension of λProlog *supports higher-order abstract syntax*
- Generic inference/reasoning step after semantics construction
- Goal: Use it for semantic/pragmatic analysis

Example: Discard wrong readings in controlled natural language

$$\begin{array}{lll} & \nearrow \mathrm{AST}_1 \longrightarrow & \cancel{\lambda x. E_{\mathsf{kin}}(x, \mathrm{quant}(2, \textbf{milli}\cdot\textbf{Newton}))} \\ \textit{``a kinetic energy of 12mN''} & & \\ & \searrow \mathrm{AST}_2 \longrightarrow & \lambda x. E_{\mathsf{kin}}(x, \mathrm{quant}(2, \textbf{meter}\cdot\textbf{Newton})) \end{array}$$

- Extension of λProlog *supports higher-order abstract syntax*
- Generic inference/reasoning step after semantics construction
- G

Examp

In [20]:
```
parse "the ball has a mass of 5 k g and a kinetic energy of 12 m N" |
        construct
```

(mass theball (quant 5 kilo gram)) ∧ (ekin theball (quant 12 milli Newton))
(mass theball (quant 5 kilo gram)) ∧ (ekin theball (quant 12 meter·Newton))

In [21]:
```
parse "the ball has a mass of 5 k g and a kinetic energy of 12 m N" |
        construct | filter -predicate=filter_pred
```

(mass theball (quant 5 kilo gram)) ∧ (ekin theball (quant 12 meter·Newton))

Example: "pairwise disjoint"

"A, B and C are pairwise disjoint"
$\mathrm{disjoint}(A, B) \wedge \mathrm{disjoint}(A, C) \wedge \mathrm{disjoint}(B, C)$

- **Approach 1**
  Semantics construction with lots of λs: *difficult!*
  $$\mathrm{disjoint}(A, B) \wedge \mathrm{disjoint}(A, C) \wedge \top \wedge \mathrm{disjoint}(B, C) \wedge \top \wedge \top \wedge \top$$
  Simplify with ELPI:
  $$\mathrm{disjoint}(A, B) \wedge \mathrm{disjoint}(A, C) \wedge \mathrm{disjoint}(B, C)$$

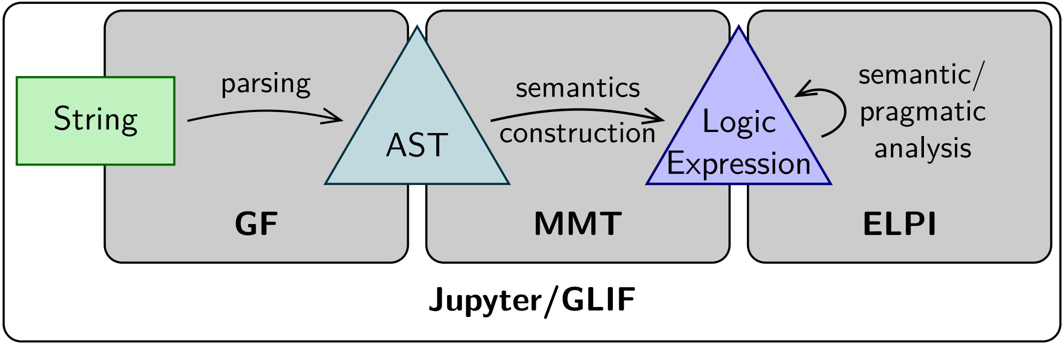
Example: “pairwise disjoint”

“A, B and C are pairwise disjoint”

$\text{disjoint}(A, B) \wedge \text{disjoint}(A, C) \wedge \text{disjoint}(B, C)$

- **Approach 1**
  Semantics construction with lots of λs: *difficult!*
  $\text{disjoint}(A, B) \wedge \text{disjoint}(A, C) \wedge \top \wedge \text{disjoint}(B, C) \wedge \top \wedge \top \wedge \top$
  Simplify with ELPI:
  $\text{disjoint}(A, B) \wedge \text{disjoint}(A, C) \wedge \text{disjoint}(B, C)$
- **Approach 2**
  Semantics construction creates preliminary expression:
  `relNT disjoint (cons A (cons B (cons C nil)))`
  Convert with ELPI: *easier*
  $\text{disjoint}(A, B) \wedge \text{disjoint}(A, C) \wedge \text{disjoint}(B, C)$

The “Normal” GLIF Pipeline



Example: Epistemic Q&A

John knows that Mary or Eve knows that Ping has a dog. (S_1)
Mary doesn't know if Ping has a dog. (S_2)
Does Eve know if Ping has a dog? (Q)

$$S_1 = \Box_{john}(\Box_{mary} hd(ping) \vee \Box_{eve} hd(ping))$$
$$S_2 = \neg(\Box_{mary} hd(ping) \vee \Box_{mary} \neg hd(ping))$$
$$Q = \Box_{eve} hd(ping) \vee \Box_{eve} \neg hd(ping)$$

$S_1, S_2 \vdash_{S5_n} Q$	$\rightsquigarrow$	yes
$S_1, S_2 \vdash_{S5_n} \neg Q$	$\rightsquigarrow$	no
else	$\rightsquigarrow$	maybe

Example: Epistemic Q&A

John knows that Mary or Eve knows that Ping has a dog. (S_1)
Mary doesn't know if Ping has a dog. (S_2)
Does Eve know if Ping has a dog? (Q)

```
p -cat=QSeq "John knows that Mary has a dog . does Mary have a dog ?" | construct -elpi
```

```
Yes!
```

```
p -cat=QSeq "Mary has a dog . does John know that Mary has a dog ?" | construct -elpi |
```

```
Maybe...
```

$$
\begin{array}{lll}
S_1, S_2 \vdash_{S5_n} Q & \rightsquigarrow & \text{yes} \\
S_1, S_2 \vdash_{S5_n} \neg Q & \rightsquigarrow & \text{no} \\
\text{else} & \rightsquigarrow & \text{maybe}
\end{array}
$$

Example: Translation

- Two German words for *"cousin"*, depending on the gender
- Two entries in abstract syntax: `cousin_female` and `cousin_male`
- Use inference to discard ASTs

"Kim is Ahmed's cousin and the father of Grace" → AST_1

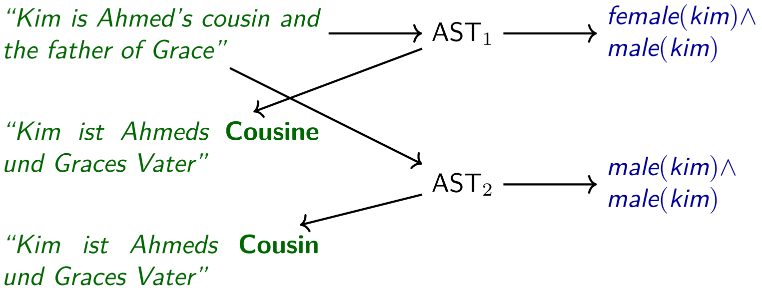
"Kim ist Ahmeds ***Cousine*** *und Graces Vater"*

AST_2

"Kim ist Ahmeds ***Cousin*** *und Graces Vater"*

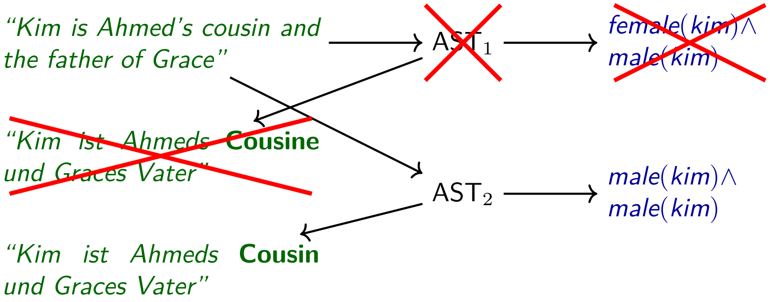
Example: Translation

- Two German words for *“cousin”*, depending on the gender
- Two entries in abstract syntax: `cousin_female` and `cousin_male`
- Use inference to discard ASTs



Example: Translation

- Two German words for *"cousin"*, depending on the gender
- Two entries in abstract syntax: `cousin_female` and `cousin_male`
- Use inference to discard ASTs



Example: Translation

- Two German words for *"cousin"*, depending on the gender
- Two entries in abstract syntax: `cousin_female` and `cousin_male`
- Use inference to discard ASTs

```
-- GLIF keeps track of the ASTs, so we can linearize the remaining readings into German:
parse -lang=Eng "Kim is Ahmed's cousin and the father of Grace" | construct |
    filter -notc -predicate=check | linearize -lang=Ger
```

Kim ist Ahmeds Cousin und Graces Vater

```
-- With all this effort we removed one of the translations we would have gotten without filtering:
parse -lang=Eng "Kim is Ahmed's cousin and the father of Grace" | linearize -lang=Ger
```

Kim ist Ahmeds Cousine und Graces Vater
Kim ist Ahmeds Cousin und Graces Vater

male(kim)

"Kim ist Ahmeds **Cousin** *und Graces Vater"*

Prover Generation

- Can describe inference rules in MMT
- $\vdash$X is the type of proofs of X *"Judgments as types"*
- Example rules:

  andEl : {A,B} $\vdash$ A∧B $\longrightarrow$ $\vdash$ A

  andEr : {A,B} $\vdash$ A∧B $\longrightarrow$ $\vdash$ B

  contra : {X} $\vdash$ male(X) $\longrightarrow$ $\vdash$ fem(X) $\longrightarrow$ $\lightning$

Prover Generation

- Can describe inference rules in MMT
- $\vdash$`X` is the type of proofs of `X` *"Judgments as types"*
- Example rules:

```
andEl  : {A,B} ⊢A∧B → ⊢A
andEr  : {A,B} ⊢A∧B → ⊢B
contra : {X} ⊢male(X) → ⊢fem(X) → ↯
```

- Generate Prolog/ELPI rules:

```
provable(A)     :- provable(A∧B).
provable(B)     :- provable(A∧B).
contradiction() :- provable(male(X)), provable(fem(X)).
```

- Extra predicates to guide proof search *iterative deepening, ...*

Prover Generation

- Can describe inference rules in MMT

```
-- generate a prover from the `calculus` theory
elpigen -mode=simpleprover calculus
-- generate signature of DDT
elpigen DDT
```

Successfully created calculus.elpi

Successfully created DDT.elpi

```
elpi-notc: checker
accumulate calculus.  % generated prover
accumulate Grammar.   % signature of ASTs (we don't use them here)
accumulate DDT.       % signature of discourse domain theory

% The `check` predicate fails if the prover found a contradiction (using iterative deepening up to depth 7)
check Item :- glif.getLog Item P, ded/hyp _ P => contradiction (idcert 7), !, fail.
check _.
```

Successfully imported checker.elpi
checker.elpi is the new default file for ELPI commands

- Extra predicates to guide proof search *iterative deepening, ...*

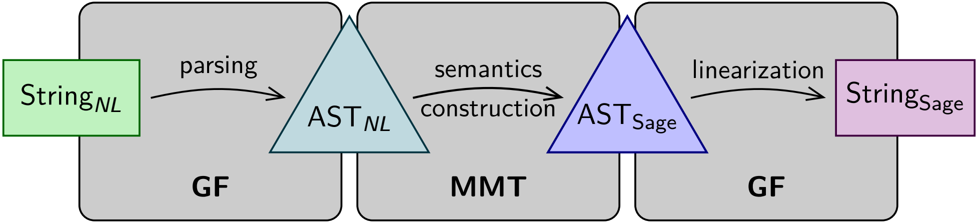
Example: Input Language for SageMath

- Can we make a natural input language for SageMath? *WolframAlpha-like*

```
sage: g = AlternatingGroup(5)
sage: g.cardinality()
60
```

"Let G be the alternating group on 5 symbols. What is the cardinality of G?"

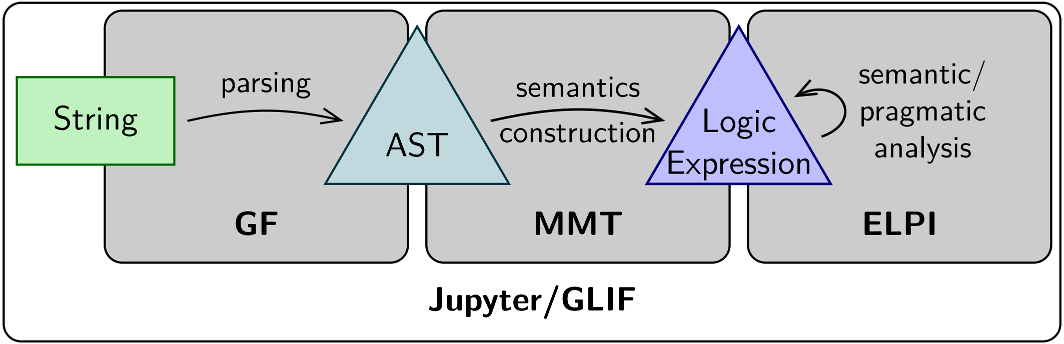
Example: Input Language for SageMath



Example: Input Language for SageMath

```
> Let G be the alternating group on 5 symbols.
# G = AlternatingGroup(5)

> Let |H| be a notation for the cardinality of H.
# def bars(H): return H.cardinality()

> What is |G|?
# print(bars(G))
60

> Let A_N be a notation for the alternating group on N symbols.
# def A(N): return AlternatingGroup(N)

> What are the cardinalities of A_4 and A_5?
# print(A(4).cardinality()); print(A(5).cardinality())
12
60
```

The “Normal” GLIF Pipeline



Example: Tableaux Machine [KK03]

- Can use tableaux for model generation
- Tableau machine: pick "best" branch as model and continue there with next sentence *like a human?*

"Ahmed or Berta paints"

$$
\begin{array}{ccc}
 & p(a) \vee p(b)^T & \\
 \swarrow & & \searrow \\
p(a)^T & & p(b)^T
\end{array}
$$

Example: Tableaux Machine [KK03]

- Can use tableaux for model generation
- Tableau machine: pick "best" branch as model and continue there with next sentence *like a human?*

"Ahmed or Berta paints"

$$
\begin{array}{ccc}
 & p(a) \vee p(b)^T & \\
 \swarrow & & \searrow \\
p(a)^T & & p(b)^T \\
| & & \\
\neg p(a)^T & & \\
| & & \\
p(a)^F & & \\
| & & \\
\bot & &
\end{array}
$$

"Ahmed doesn't paint"

Example: Tableaux Machine [KK03]

- Can use tableaux for model generation
- Tableau machine: pick "best" branch as model and continue there with next sentence *like a human?*

"Ahmed or Berta paints"

"Ahmed doesn't paint"

$$\begin{array}{ccc} & p(a) \vee p(b)^T & \\ p(a)^T & & p(b)^T \\ | & & | \\ \neg p(a)^T & & \neg p(a)^T \\ | & & | \\ p(a)^F & & p(a)^F \\ | & & \\ \bot & & \end{array}$$

Example: Tableaux Machine

Background Knowledge

$\forall x. fem(x) \Rightarrow \neg masc(x)$
$masc(j)$
$fem(m)$

$\downarrow$

$talkto(j, m)$

$\downarrow$

$sad(s)$

"John talks to Mary."

$talkto(j, m)$

"Sasha is sad."

$sad(s)$

Example: Tableaux Machine

Background Knowledge

$\forall x. fem(x) \Rightarrow \neg masc(x)$
$masc(j)$
$fem(m)$

$\downarrow$

$talkto(j, m)$

$\downarrow$

$sad(s)$

"John talks to Mary."

$talkto(j, m)$

"Sasha is sad."

$sad(s)$

"He loves her."

$\exists X. masc(X) \wedge$
$\quad \exists Y. fem(Y) \wedge love(X, Y)$

$love(s, s)$	$love(s, m)$
$masc(s)$	$masc(s)$
$fem(s)$	
$\perp$	

Example: Tableaux Machine

Background Knowledge

"John talks to Mary."

$talkto(j, m)$

"Sasha is sad."

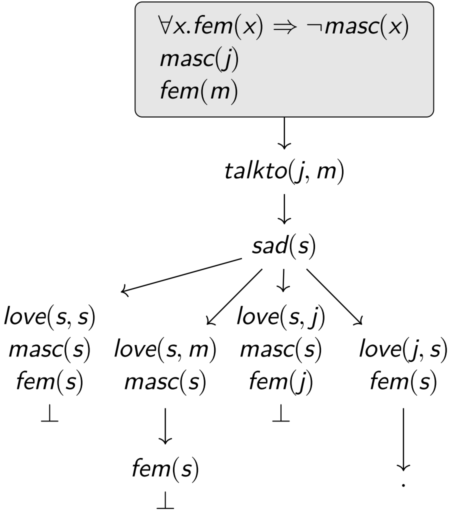
$sad(s)$

"He loves her."

$\exists X.masc(X) \wedge$
$\quad \exists Y.fem(Y) \wedge love(X, Y)$

"Sasha is a woman."

$fem(s)$



Example: Tableaux Machine

Background Knowledge

"John talks to Mary."

$talkto(j, m)$

"Sasha is sad."

$sad(s)$

"He loves her."

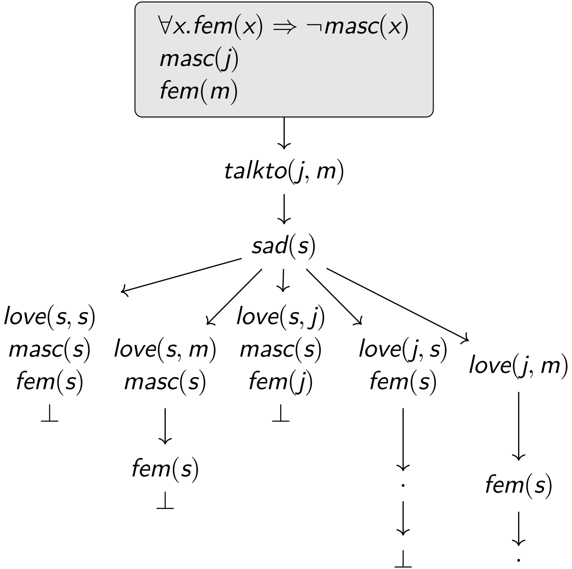
$\exists X.masc(X) \wedge$
$\quad \exists Y.fem(Y) \wedge love(X, Y)$

"Sasha is a woman."

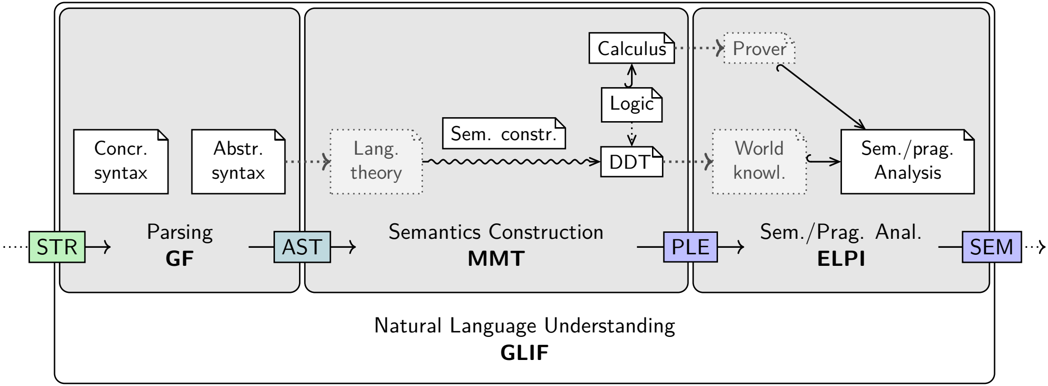
$fem(s)$

"John doesn't love Sasha."

$\neg love(j, s)$

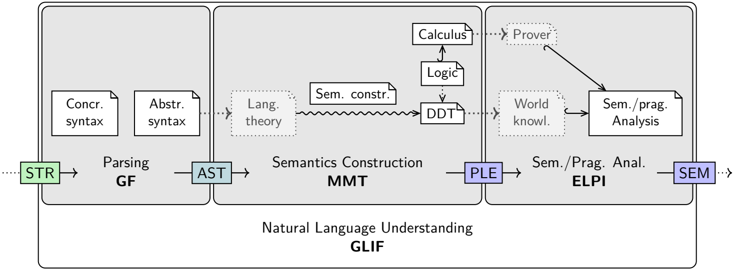


Summary: A GLIF Specification



Conclusion

- GLIF = GF + MMT + ELPI
- Prototyping natural language understanding
- Access via Jupyter notebooks
- We use it for teaching



Natural Deduction in MMT/LF

$$\frac{A \wedge B}{A} \wedge El \qquad\qquad \frac{A \vee B \quad \begin{array}{c}[A]^1\\ \vdots \\ C\end{array} \quad \begin{array}{c}[B]^1\\ \vdots \\ C\end{array}}{C} \vee E^1$$

```
// ⊢X is type of proofs for X (judgments as types)
⊢ : o → type

∧El : Π_{A:o} Π_{B:o} ⊢A∧B → ⊢A
∨E  : Π_{A:o} Π_{B:o} Π_{C:o} ⊢A∨B → (⊢A → ⊢C) → (⊢B → ⊢C) → ⊢C
```

Generating Provers in ELPI

LF rule $\wedge \texttt{El} : \Pi_{\texttt{A:o}} \Pi_{\texttt{B:o}} \vdash \texttt{A} \wedge \texttt{B} \longrightarrow \vdash \texttt{A}$

ELPI equivalent

direct: `pi A \ pi B \ ded (and A B) => ded A.`

syn. sugar: `ded A :- ded (and A B).`

LF rule $\wedge\mathtt{El} : \Pi_{\mathtt{A:o}}\Pi_{\mathtt{B:o}}\ \vdash\mathtt{A}\wedge\mathtt{B} \to \vdash\mathtt{A}$

ELPI equivalent

direct: `pi A \ pi B \ ded (and A B) => ded A.`

syn. sugar: `ded A :- ded (and A B).`

Example: Or-Elimination

LF: $\vee\mathtt{E} : \Pi_{\mathtt{A:o}}\Pi_{\mathtt{B:o}}\Pi_{\mathtt{C:o}}\ \vdash\mathtt{A}\vee\mathtt{B} \to (\vdash\mathtt{A} \to \vdash\mathtt{C}) \to (\vdash\mathtt{B} \to \vdash\mathtt{C}) \to \vdash\mathtt{C}$

ELPI: `ded C :- ded (or A B), ded A => ded C, ded B => ded C.`

Example: Forall-Introduction

LF: $\forall\mathtt{I} : \Pi_{\mathtt{P}:\iota \to \mathtt{o}}\ (\Pi_{\mathtt{x}:\iota}\ \vdash\mathtt{P\ x}) \to \vdash\forall\mathtt{P}$

ELPI: `ded (forall P) :- pi x \ ded (P x).`

Controlling the Proof Search

- Problem: Search diverges *searching harder than checking*
- Solution: Control search with helper predicates:
  inspired by ProofCert project by Miller et al.
  - Intuition: Decide whether to apply rule
  - Do not affect correctness
  - Extra argument tracks aspects of proof state

Before: `ded A :- ded (and A B).`

Now: `ded X A :- help/andEl X A B X1, ded X1 (and A B).`

Helper Predicates

Name	Predicate	Argument
Iter. deepening	checks depth	remaining depth
Proof term	generates term	proof term
Product	calls other predicates	arguments for other predicates
Backchaining	Prolog's backchaining ($\approx$ forward reasoning from axioms via $\Rightarrow/\forall$ elimination rules)	pattern of formula to be proven (e.g. a conjunction)

Example helper: Iterative deepening

```
help/andEl (idcert N) _ _ (idcert N1) :- N > 0, N1 is N - 1.
```

Tableau Provers

$$\frac{A \wedge B^F}{A^F \mid B^F} \wedge^F \qquad\qquad \dfrac{A \wedge B^F \quad \begin{matrix}[A^F] \\ \vdots \\ \bot\end{matrix} \quad \begin{matrix}[B^F] \\ \vdots \\ \bot\end{matrix}}{\bot} \wedge^F$$

LF: $\wedge^{\boldsymbol{F}} : \Pi_{\mathtt{A:o}}\Pi_{\mathtt{B:o}}\ \mathtt{A{\wedge}B}^{\boldsymbol{F}} \to (\mathtt{A}^{\boldsymbol{F}} \to \bot) \to (\mathtt{B}^{\boldsymbol{F}} \to \bot) \to \bot$

ELPI:
```
closed X :- help/andF X A B X1 X2 X3, f X1 (and A B),
                        f/hyp A => closed X2, f/hyp B => closed X3.
```

With iterative deepening we get a working prover!
$\to$ Other helpers result in more efficient provers

References I

[KK03] Michael Kohlhase and Alexander Koller. "Resource-Adaptive Model Generation as a Performance Model". In: *Logic Journal of the IGPL* 11.4 (2003), pp. 435–456. URL: http://jigpal.oxfordjournals.org/cgi/content/abstract/11/4/435.

[Mon70] R. Montague. "English as a Formal Language". In: Reprinted in [Tho74], 188–221. Edizioni di Communita, Milan, 1970, pp. 189–224.

[Mon74] Richard Montague. "The Proper Treatment of Quantification in Ordinary English". In: *Formal Philosophy. Selected Papers*. Ed. by R. Thomason. New Haven: Yale University Press, 1974.

[Tho74] R. Thomason, ed. *Formal Philosophy: selected Papers of Richard Montague*. Yale University Press, New Haven, CT, 1974.