Logical Syntax and Semantics

A logical language based on:

1. function-argument structures: \((M \, N)\)
2. lambda abstraction: \(\lambda x. (\alpha \, x)\)
3. beta-reduction: \((\lambda x.(M \, x) \, N) \equiv (M \, N)\)
4. Boolean combinations: \((\phi \land \psi), \ldots\)
5. Quantified formulas: \(\forall x. \phi, \exists x. \phi\)

Models for the language:

- \(M = (D, V)\)
- variable assignment \(g : \text{Var} \rightarrow D\)
- recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\)
- \(M, g \models \phi \text{ if } [\phi]^{M,g} = 1\)
Logical Syntax and Semantics

- A logical language based on:
  1. function-argument structures: \((M N)\)
  2. lambda abstraction: \(\lambda x.(\alpha x)\)
  3. beta-reduction: \((\lambda x.(M x) N) \equiv (M N)\)
  4. Boolean combinations: \((\phi \land \psi), \ldots\)
  5. Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

- Models for the language:
  - \(M = (D, V)\)
  - variable assignment \(g : \text{Var} \rightarrow D\)
  - recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\)
  - \(M, g \models \phi \iff [\phi]^{M,g} = 1\)
Logical Syntax and Semantics

A logical language based on:

- function-argument structures: \((MN)\)
- lambda abstraction: \(\lambda x.(\alpha x)\)
- beta-reduction: \((\lambda x.(M x) N) \equiv (MN)\)
- Boolean combinations: \((\phi \land \psi), \ldots\)
- Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

Models for the language:

- \(M = (D, V)\)
- variable assignment \(g : \text{Var} \rightarrow D\)
- recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\)
- \(M, g \models \phi \iff [\phi]^{M,g} = 1\)
Logical Syntax and Semantics

A logical language based on:

1. function-argument structures: \((M N)\)
2. lambda abstraction: \(\lambda x.(\alpha x)\)
3. beta-reduction: \((\lambda x.(M x) N) \equiv (M N)\)
4. Boolean combinations: \((\phi \land \psi), \ldots\)
5. Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

Models for the language:

1. \(M = (D, V)\)
2. variable assignment \(g : \text{Var} \rightarrow D\)
3. recursive definition of \(\llbracket \alpha \rrbracket^M_g\) for expressions \(\alpha\)
4. \(M, g \models \phi \iff \llbracket \phi \rrbracket^M_g = 1\)
Logical Syntax and Semantics

- A logical language based on:
  1. function-argument structures: $(M \ N)$
  2. lambda abstraction: $\lambda x.(\alpha \ x)$
  3. beta-reduction: $(\lambda x.(M \ x) \ N) \equiv (M \ N)$
  4. Boolean combinations: $(\phi \land \psi)$, ...
  5. Quantified formulas: $\forall x.\phi$, $\exists x.\phi$

- Models for the language:
  1. $M = (D, V)$
  2. variable assignment $g : Var \rightarrow D$
  3. recursive definition of $[\alpha]^M_g$ for expressions $\alpha$.
Logical Syntax and Semantics

A logical language based on:

1. function-argument structures: 
   \((M \ N)\)
2. lambda abstraction: 
   \(\lambda x.(\alpha x)\)
3. beta-reduction: 
   \((\lambda x.(M \ x) \ N) \equiv (M \ N)\)
4. Boolean combinations: 
   \((\phi \land \psi), \ldots\)
5. Quantified formulas: 
   \(\forall x.\phi, \exists x.\phi\)

Models for the language:

\[ M = (D, V) \]
\[ \text{variable assignment } g : \text{Var} \rightarrow D \]
\[ \text{recursive definition of } [\alpha]^{M,g} \text{ for expressions } \alpha. \]
\[ M, g \models x \iff [\phi]^{M,g} = 1. \]
A logical language based on:

1. function-argument structures: \((M N)\)
2. lambda abstraction: \(\lambda x.(\alpha x)\)
3. beta-reduction: \((\lambda x.(M x) N) \equiv (M N)\)
4. Boolean combinations: \((\phi \land \psi), \ldots\)
5. Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

Models for the language:

1. \(M = \langle D, V \rangle\)
2. variable assignment \(g : \text{Var} \rightarrow D\)
3. recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\).
4. \(M, g \models \phi\) iff \([\phi]^{M,g} = 1\).
A logical language based on:

1. function-argument structures: \((M N)\)
2. lambda abstraction: \(\lambda x.(\alpha x)\)
3. beta-reduction: \((\lambda x.(M x) N) \equiv (M N)\)
4. Boolean combinations: \((\phi \land \psi), \ldots\)
5. Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

Models for the language:

1. \(M = \langle D, V \rangle\)
2. variable assignment \(g : \text{Var} \mapsto D\)
3. recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\).
4. \(M, g \models \phi\) iff \([\phi]^{M,g'} = 1\).
Logical Syntax and Semantics

- A logical language based on:
  1. function-argument structures: \((M \ N)\)
  2. lambda abstraction: \(\lambda x. (\alpha \ x)\)
  3. beta-reduction: \((\lambda x. (M \ x) \ N) \equiv (M \ N)\)
  4. Boolean combinations: \((\phi \land \psi), \ldots\)
  5. Quantified formulas: \(\forall x. \phi, \exists x. \phi\)

- Models for the language:
  1. \(M = \langle D, V \rangle\)
  2. variable assignment \(g : Var \rightarrow D\)
  3. recursive definition of \([\alpha]_{M,g}^{M}\) for expressions \(\alpha\).
  4. \(M, g \models \phi\) iff \([\phi]_{M,g'} = 1\).
Logical Syntax and Semantics

A logical language based on:

1. function-argument structures: \((M N)\)
2. lambda abstraction: \(\lambda x.(\alpha x)\)
3. beta-reduction: \((\lambda x.(M x) N) \equiv (M N)\)
4. Boolean combinations: \((\phi \land \psi), \ldots\)
5. Quantified formulas: \(\forall x.\phi, \exists x.\phi\)

Models for the language:

1. \(M = \langle D, V \rangle\)
2. variable assignment \(g : Var \rightarrow D\)
3. recursive definition of \(\llbracket \alpha \rrbracket_{M,g}\) for expressions \(\alpha\).
4. \(M, g \models \phi\) iff \(\llbracket \phi \rrbracket_{M,g'} = 1\).
Logical Syntax and Semantics

- A logical language based on:
  1. function-argument structures: \((M N)\)
  2. lambda abstraction: \(\lambda x. (\alpha x)\)
  3. beta-reduction: \((\lambda x. (M x) N) \equiv (M N)\)
  4. Boolean combinations: \((\phi \land \psi), \ldots\)
  5. Quantified formulas: \(\forall x. \phi, \exists x. \phi\)

- Models for the language:
  1. \(M = \langle D, V \rangle\)
  2. variable assignment \(g : Var \mapsto D\)
  3. recursive definition of \([\alpha]^{M,g}\) for expressions \(\alpha\).
  4. \(M, g \models \phi\) iff \([\phi]^{M,g'} = 1\).
Compositional Semantics

**Compositionality** The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the **parts** are?

- Feature-based context-free grammar formalism.
- Every category has a `sem` feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF
  - phrasal categories: build an LF by function application over the LFs of the daughters

Example PS Rule

\[ S[sem = \langle app(?subj,?vp)\rangle] \rightarrow NP[sem=?subj] \, VP[sem=?vp] \]
Compositionality  The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the parts are?

- Feature-based context-free grammar formalism.
- Every category has a \textit{sem} feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF.
  - phrasal categories: build an LF by function application over the LFs of the daughters.

**Example PS Rule**

\[ S[\text{sem} = \text{app (?)subj, ?vp}] \rightarrow \text{NP}[\text{sem} = \text{?subj}] \text{ VP}[\text{sem} = \text{?vp}] \]
**Compositionality** The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the **parts** are?

- Feature-based context-free grammar formalism.
- Every category has a **sem** feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF.
  - phrasal categories: build an LF by function application over the LFs of the daughters.

**Example PS Rule**

\[
S[\text{sem} = \text{app}(?\text{subj}, ?\text{vp})] \rightarrow \text{NP}[\text{sem}=?\text{subj}] \text{ VP}[\text{sem}=?\text{vp}]
\]
Compositional Semantics

**Compositionality** The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the parts are?

- Feature-based context-free grammar formalism.
- Every category has a `sem` feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF.
  - phrasal categories: build an LF by function application over the LFs of the daughters.

**Example PS Rule**

\[ S[\text{sem} = \langle \text{app}(\text{?subj}, \text{?vp}) \rangle] \rightarrow \text{NP}[\text{sem} = \text{?subj}] \text{ VP}[\text{sem} = \text{?vp}] \]
Compositional Semantics

**Compositionality**  The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the parts are?

- Feature-based context-free grammar formalism.
- Every category has a `sem` feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF.
  - phrasal categories: build an LF by function application over the LFs of the daughters.

**Example PS Rule**

\[
S[\text{sem} = \langle \text{app}(\text{?subj}, \text{?vp}) \rangle] \rightarrow \text{NP}[\text{sem} = \text{?subj}] \ \text{VP}[\text{sem} = \text{?vp}]
\]
Compositional Semantics

**Compositionality** The meaning of a complex expression is a function of the meaning of its parts.

How do we know what the parts are?

- Feature-based context-free grammar formalism.
- Every category has a **sem** feature whose value is the semantics of expressions of that category:
  - lexical categories: fully-instantiated LF.
  - phrasal categories: build an LF by function application over the LFs of the daughters.

**Example PS Rule**

\[ S[\text{sem} = \langle \text{app}(?,?) \rangle] \rightarrow NP[\text{sem}=?] \ VP[\text{sem}=?] \]
Computational Recap

- Logical expressions are parsed into subclasses of Expression by `nltk.sem.logic`.
- Expressions can be evaluated in a model by `nltk.sem.evaluate`.
- English sentences can be parsed into LF by `nltk.parse.featurechart (via the nltk.parse.load_earley() function.)`

Sample Interpretation

\[ A \text{ dog barks} \quad \exists x.((\text{dog } x) \land (\text{bark } x)) \quad \rightarrow \]
\[ \llbracket \exists x.((\text{dog } x) \land (\text{bark } x)) \rrbracket_{M,g}^{g} = 1 \text{ iff } \ldots \]
Logical expressions are parsed into subclasses of Expression by `nltk.sem.logic`.

Expressions can be evaluated in a model by `nltk.sem.evaluate`.

English sentences can be parsed into LF by `nltk.parse.featurechart` (via the `nltk.parse.load_earley()` function.)

Sample Interpretation

A dog barks  
$\exists x.((\text{dog } x) \land (\text{bark } x)) \quad \rightarrow$  
$[\exists x.((\text{dog } x) \land (\text{bark } x))]^{M,g} = 1 \text{ iff } \ldots$
Computational Recap

- Logical expressions are parsed into subclasses of `Expression` by `nltk.sem.logic`.
- Expressions can be evaluated in a model by `nltk.sem.evaluate`.
- English sentences can be parsed into LFs by `nltk.parse.featurechart` (via the `nltk.parse.load_earley()` function.)

Sample Interpretation

\[
\text{A dog barks} \quad \longrightarrow \\
\exists x. ((\text{dog } x) \land (\text{bark } x)) \quad \longrightarrow \\
[\exists x. ((\text{dog } x) \land (\text{bark } x))]^{M,g} = 1 \text{ iff ...}
\]
Computational Recap

- Logical expressions are parsed into subclasses of `Expression` by `nltk.sem.logic`.
- Expressions can be evaluated in a model by `nltk.sem.evaluate`.
- English sentences can be parsed into LFs by `nltk.parse.featurechart` (*via* the `nltk.parse.load_earley()` function.)

Sample Interpretation

\[ \exists x. (\text{dog } x \land \text{bark } x) \]

\[ M, g = 1 \text{iff...} \]
import nltk
tokens = 'a dog barks'.split()
from nltk.parse import load_earley
cp = load_earley('grammars/sem1.fcfg', trace=0)
trees = cp.nbest_parse(tokens)
for t in trees:
    print t
Parse for *A dog barks*

\[
(S[sem=<\text{some } x. (\text{and} (\text{dog } x) (\text{bark } x))>]
(NP[sem=<\text{P. some } x. (\text{and} (\text{dog } x) (\text{P } x))>]
(Det[sem=<\text{Q P. some } x. (\text{and} (\text{Q } x) (\text{P } x))>] a)
(N[sem=<\text{dog}>] \text{dog})
(VP[sem=<\text{x. (bark } x)>]
(IV[sem=<\text{x. (bark } x)>] \text{barks})))
\]
from nltk.sem import *
val = Valuation({
    'fido': 'f',
    'dog': {'f': True, 'd': True},
    'bark': {'d': True},
})
dom = val.domain
m = Model(dom, val)
g = Assignment(dom)
Truth in model m

```python
>>> print m
Domain = set(['d', 'f']),
Valuation =
'bark': {'d': True},
'dog': {'d': True, 'f': True},
'fido': 'f'
>>> g
{}
>>> m.evaluate('some x. ((dog x) and (bark x))', g)
True
```
Tracing

Truth in model \( m \)

\[
>>> m.\text{evaluate}(\text{'some } x. (\text{dog } x) \text{ and (bark } x))', g, \text{trace}=1)
\]

Open formula is '(and (dog x) (bark x))' with assignment \( g \)
(trying assignment \( g[d/x] \))
value of '(and (dog x) (bark x))' under \( g[d/x] \) is True
(trying assignment \( g[f/x] \))
value of '(and (dog x) (bark x))' under \( g[f/x] \) is False
'(and (dog x) (bark x))' evaluates to True under \( M, g \)
'some x. ((dog x) and (bark x))' evaluates to True under \( M \).
from nltk.sem import *  
val = Valuation({
    'fido': 'f',
    'kim': 'k',
    'chase': {'f': {'k': True},
              'k': {'f': True}}
})

dom = val.domain
m = Model(dom, val)
g = Assignment(dom)
from nltk.sem import *
val = Valuation()
v = [('fido', 'f'),
     ('kim', 'k'),
     ('chase', set([('f', 'k'), ('k', 'f')]))
]
val.read(v)

dom = val.domain
m = Model(dom, val)
g = Assignment(dom)
from nltk.sem import *

v = ""
    fido => f
    kim => k
    chase => {(f, k), (k, f)}
"

val = parse_valueation(v)
dom = val.domain
m = Model(dom, val)
g = Assignment(dom)
Examining Valuations

Outputting tuples

```python
>>> val
{'f': 'f', 'kim': 'k',
 'chase': {'k': {'f': True}, 'f': {'k': True}}}
>>> relation = val['chase']
>>> relation
{'k': {'f': True}, 'f': {'k': True}}
>>> relation.tuples()
set([('k', 'f'), ('f', 'k')])
>>> val['run']
Traceback (most recent call last):
 ...
ltk.sem.evaluate.Undefined: Unknown expression: 'run'
>>> m.evaluate('\x. (chase x kim)', g)
{'f': True}
>>> m.evaluate('\x. some y. (chase x y)', g).tuples()
set([('k', 'f')])
```
Parse sentence & load valuation

```python
from nltk.parse import FeatureEarleyChartParser
import nltk.data
grammar = nltk.data.load('grammars/sem2.fcfg')
val = nltk.data.load('grammars/valuation1.val')
dom = val.domain
m = Model(dom, val)
g = Assignment(dom)
sent = 'some girl chases a dog'
result = nltk.sem.text_evaluate([sent], grammar, m, g)
for (syntree, semrep, value) in result[sent]:
    print """%s' is %s in Model m\n""" % (semrep.infixify(), value)
```
Result

'some x.((girl x) and
  some z559.((dog z559) and
    (chase z559 x)))'

is True in Model m
The NLTK implementation yields an end-to-end mapping:

- Compute all parses of a sentence $S$ relative to a feature-based CFG;
- provide a logical form for each constituent of $S$;
- parse the logical form LF for each reading of $S$;
- build a representation of a first order model $M$;
- recursively evaluate LF in $M$.
- If LF contains free variables, then value also depends on $g$.

Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

Two approaches in nltk.contrib: hole.py and glue.semantics package.
Summary

- The NLTK implementation yields an end-to-end mapping:
  - Compute all parses of a sentence $S$ relative to a feature-based CFG;
  - provide a logical form for each constituent of $S$;
  - parse the logical form LF for each reading of $S$;
  - build a representation of a first order model $M$;
  - recursively evaluate LF in $M$.
  - If LF contains free variables, then value also depends on $g$.

- Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

- Two approaches in `nltk.contrib: hole.py` and `gluesemantics` package.
Summary

- The NLTK implementation yields an end-to-end mapping:
  - Compute all parses of a sentence $S$ relative to a feature-based CFG;
  - provide a logical form for each constituent of $S$;
  - parse the logical form LF for each reading of $S$;
  - build a representation of a first order model $M$;
  - recursively evaluate LF in $M$.
  - If LF contains free variables, then value also depends on $g$.

- Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

- Two approaches in nltk.contrib: hole.py and gluesemantics package.
Summary

- The NLTK implementation yields an end-to-end mapping:
  - Compute all parses of a sentence $S$ relative to a feature-based CFG;
  - provide a logical form for each constituent of $S$;
  - parse the logical form LF for each reading of $S$;
  - build a representation of a first order model $M$;
  - recursively evaluate LF in $M$.
  - If LF contains free variables, then value also depends on $g$.

- Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

- Two approaches in nltk.contrib: hole.py and glue semantics package.
The NLTK implementation yields an end-to-end mapping:

- Compute all parses of a sentence $S$ relative to a feature-based CFG;
- provide a logical form for each constituent of $S$;
- parse the logical form LF for each reading of $S$;
- build a representation of a first order model $M$;
- recursively evaluate LF in $M$.
- If LF contains free variables, then value also depends on $g$.

Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

Two approaches in nltk.contrib: hole.py and glue.semantics package.
Summary

- The NLTK implementation yields an end-to-end mapping:
  - Compute all parses of a sentence $S$ relative to a feature-based CFG;
  - provide a logical form for each constituent of $S$;
  - parse the logical form LF for each reading of $S$;
  - build a representation of a first order model $M$;
  - recursively evaluate LF in $M$.
  - If LF contains free variables, then value also depends on $g$.

- Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

- Two approaches in nltk.contrib: hole.py and glue semantics package.
The NLTK implementation yields an end-to-end mapping:

- Compute all parses of a sentence $S$ relative to a feature-based CFG;
- provide a logical form for each constituent of $S$;
- parse the logical form LF for each reading of $S$;
- build a representation of a first order model $M$;
- recursively evaluate LF in $M$.
- If LF contains free variables, then value also depends on $g$.

Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

Two approaches in `nltk.contrib: hole.py` and `gluesemantics` package.
The NLTK implementation yields an end-to-end mapping:

- Compute all parses of a sentence $S$ relative to a feature-based CFG;
- provide a logical form for each constituent of $S$;
- parse the logical form LF for each reading of $S$;
- build a representation of a first order model $M$;
- recursively evaluate LF in $M$.
- If LF contains free variables, then value also depends on $g$.

Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

Two approaches in nltk.contrib: hole.py and gluesemantics package.
The NLTK implementation yields an end-to-end mapping:

- Compute all parses of a sentence $S$ relative to a feature-based CFG;
- provide a logical form for each constituent of $S$;
- parse the logical form LF for each reading of $S$;
- build a representation of a first order model $M$;
- recursively evaluate LF in $M$.
- If LF contains free variables, then value also depends on $g$.

Major shortcoming so far: no treatment of semantic ambiguity, e.g., quantifier scope ambiguity.

Two approaches in nltk.contrib: hole.py and glue-semantics package.