A Model of a Fragment of English, Part 2

Kenneth Lai

Brandeis University

October 13, 2022

Announcements

- For next Tuesday
  - Read van Eijck and Unger Chapter 8
- For next Wednesday
  - HW2 due
  - Paper Presentation Ideas due

Today's Plan

- Paper Presentation Idea: Bridging Formal and Distributional Semantics
- A Model of a Fragment of English

Bridging Formal and Distributional Semantics

- Baroni and Zamparelli. 2010. Nouns are Vectors, Adjectives are Matrices: Representing Adjective-Noun Constructions in Semantic Space. Proceedings of EMNLP.
- Socher et al. 2012. Semantic Compositionality through Recursive Matrix-Vector Spaces. Proceedings of EMNLP.
- Venhuizen et al. 2022. Distributional Formal Semantics. Information and Computation, 287:104763.

- Also see CL Special Issue on Formal Distributional Semantics
- Also see Bridges and Gaps between Formal and Computational Linguistics (an ESSLLI 2022 workshop)
  - Not a source of papers, but interesting to look at nonetheless

Summary of 9/14 Discussion

Things in model	Expression	Type
relations	verbs	`String`
entities	nouns	`String`
?	adjectives	`String`
truth values	sentences	`String`

Summary of 9/14 Discussion

Things in model	Expression	Type
relations	verbs	`String`
entities	nouns	`String`
?	adjectives	`String`
truth values	sentences	`String`

- How to represent a model in Haskell?

Summary of 9/14 Discussion

Things in model	Expression	Type
relations	verbs	`String`
entities	nouns	`String`
?	adjectives	`String`
truth values	sentences	`String`

- How to represent a model in Haskell?
- Truth values (`True`, `False`) are objects of type `Bool`

A Model of a Fragment of English

- Declare a data type `Entity`

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
     deriving (Eq,Show,Bounded,Enum)
```

A Model of a Fragment of English

- Declare a data type `Entity`

```
data Entity = A | B | C | D | E | F | G
            | H | I | J | K | L | M | N
            | O | P | Q | R | S | T | U
            | V | W | X | Y | Z | Unspec
     deriving (Eq,Show,Bounded,Enum)
```

- We can put all of our entities in a list

```
entities :: [Entity]
entities =  [minBound..maxBound]
```

A Model of a Fragment of English

- Proper names are interpreted as entities

```
snowWhite, alice, dorothy, goldilocks, littleMook, atreyu
                                                    :: Entity

snowWhite  = S
alice      = A
dorothy    = D
goldilocks = G
littleMook = M
atreyu     = Y
```

A Model of a Fragment of English

- Proper names are interpreted as entities

```
snowWhite, alice, dorothy, goldilocks, littleMook, atreyu
                                                  :: Entity

snowWhite  = S
alice      = A
dorothy    = D
goldilocks = G
littleMook = M
atreyu     = Y
```

- Not all nouns are interpreted as entities, though
  - Common nouns such as *girl* and *dwarf* are more like sets of entities, or properties of entities (unary relations)

A Model of a Fragment of English

- Relations are represented as their characteristic functions
  - Given some number of entities, does the relation hold between them?

```
type OnePlacePred   = Entity -> Bool
type TwoPlacePred   = Entity -> Entity -> Bool
type ThreePlacePred = Entity -> Entity -> Entity -> Bool
```

A Model of a Fragment of English

- Relations are represented as their characteristic functions
  - Given some number of entities, does the relation hold between them?

```
type OnePlacePred   = Entity -> Bool
type TwoPlacePred   = Entity -> Entity -> Bool
type ThreePlacePred = Entity -> Entity -> Entity -> Bool
```

- Convert a list of entities into a function

```
list2OnePlacePred :: [Entity] -> OnePlacePred
list2OnePlacePred xs = \ x -> elem x xs
```

A Model of a Fragment of English

- Common nouns are interpreted as one-place predicates

```
girl, boy, princess, dwarf, giant, wizard, sword, dagger
                                              :: OnePlacePred

girl     = list2OnePlacePred [S,A,D,G]
boy      = list2OnePlacePred [M,Y]
princess = list2OnePlacePred [E]
dwarf    = list2OnePlacePred [B,R]
giant    = list2OnePlacePred [T]
wizard   = list2OnePlacePred [W,V]
sword    = list2OnePlacePred [F]
dagger   = list2OnePlacePred [X]
```

A Model of a Fragment of English

- Common nouns are interpreted as one-place predicates

```
child, person, man, woman, male, female, thing
                                   :: OnePlacePred

child  = \ x -> (girl x  || boy x)
person = \ x -> (child x || princess x || dwarf x
                         || giant x    || wizard x)
man    = \ x -> (dwarf x || giant x || wizard x)
woman  = \ x -> princess x
male   = \ x -> (man x || boy x)
female = \ x -> (woman x || girl x)
thing  = \ x -> not (person x || x == Unspec)
```

A Model of a Fragment of English

- Intransitive verbs are also interpreted as one-place predicates

```
laugh, cheer, shudder :: OnePlacePred

laugh   = list2OnePlacePred [A,G,E]
cheer   = list2OnePlacePred [M,D]
shudder = list2OnePlacePred [S]
```

A Model of a Fragment of English

- Transitive verbs are interpreted as two-place predicates

```
love, admire, help, defeat :: TwoPlacePred

love   = curry (`elem` [(Y,E),(B,S),(R,S)])
admire = curry (`elem` [(x,G) | x <- entities, person x])
help   = curry (`elem` [(W,W),(V,V),(S,B),(D,M)])
defeat = curry (`elem` [(x,y) | x <- entities,
                                y <- entities,
                                dwarf x && giant y]
                      ++ [(A,W),(A,V)])
```

A Model of a Fragment of English

- Transitive verbs are interpreted as two-place predicates

```
love, admire, help, defeat :: TwoPlacePred

love   = curry (`elem` [(Y,E),(B,S),(R,S)])
admire = curry (`elem` [(x,G) | x <- entities, person x])
help   = curry (`elem` [(W,W),(V,V),(S,B),(D,M)])
defeat = curry (`elem` [(x,y) | x <- entities,
                                y <- entities,
                                dwarf x && giant y]
                     ++ [(A,W),(A,V)])
```

- curry converts a function of a pair of arguments into a sequence of functions of one argument

```
curry :: ((a,b) -> c) -> a -> b -> c
curry f x y = f (x,y)
```

A Model of a Fragment of English

- Ditransitive verbs are interpreted as three-place predicates

```
curry3 :: ((a,b,c) -> d) -> a -> b -> c -> d
curry3 f x y z = f (x,y,z)

give :: ThreePlacePred
give = curry3 (`elem` [(T,S,X),(A,E,S)])
```

A Model of a Fragment of English

Things in model	Expression	Type
relations	verbs	`String`
entities	nouns	`String`
?	adjectives	`String`
truth values	sentences	`String`

A Model of a Fragment of English

Things in model	Expression	Type
truth values	sentences	`String`
entities	proper names	`String`
unary relations	common nouns	`String`
unary relations	intransitive verbs	`String`
binary relations	transitive verbs	`String`
ternary relations	ditransitive verbs	`String`
?	adjectives	`String`

A Model of a Fragment of English

Things in model	Expression	Type
truth values	sentences	`String`
entities	proper names	`String`
unary relations	common nouns	`String`
unary relations	intransitive verbs	`String`
binary relations	transitive verbs	`String`
ternary relations	ditransitive verbs	`String`
?	adjectives	`String`

- **Exercise** What about adjectives? (You can consider adjectives to be words that combine with common nouns to form complex noun phrases, e.g., "friendly" + "wizard" = "friendly wizard". You do not have to consider predicative uses of adjectives, e.g., "Snow White is friendly.".)

A Model of a Fragment of English

Things in model	Expression	Type
truth values	sentences	`String`
entities	proper names	`String`
unary relations	common nouns	`String`
unary relations	intransitive verbs	`String`
binary relations	transitive verbs	`String`
ternary relations	ditransitive verbs	`String`
?	adjectives	`String`

- On one level, everything is (or can be represented as) a `String`
  - `String` is not necessarily the most useful type for semantic interpretation, though

A Model of a Fragment of English

- Principle of Compositionality
  - "...the meaning of a complex expression depends on the meanings of its parts and the way they are combined syntactically."

A Model of a Fragment of English

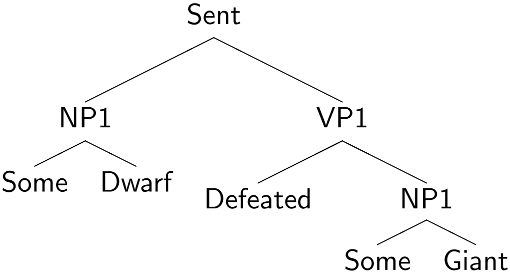
- Principle of Compositionality
  - "...the meaning of a complex expression depends on the meanings of its parts and the way they are combined syntactically."
- We want to give structure to our sentences
  - These structures will tell us how to combine the meanings of expressions to get meanings of bigger expressions

A Model of a Fragment of English

- Parsing is the process of constructing syntax data structures from strings of words
    - Take LING 120B for more details about these structures, and COSI 114B for more details about how to create them
    - Also see van Eijck and Unger Chapter 9

A Model of a Fragment of English

- In this class, we will assume that these structures are given to us




```
(Sent (NP1 Some Dwarf)
      (VP1 Defeated (NP1 Some Giant)))
```

A Model of a Fragment of English

- A computational grammar (adapted from `FSynF.hs`)

```
data Sent = Sent NP VP deriving Show
data NP   = SnowWhite  | Alice  | Dorothy | Goldilocks
          | LittleMook | Atreyu | Everyone | Someone
          | NP1 DET CN | NP2 DET RCN
          deriving Show
data DET  = The | Every | Some | No | Most
          deriving Show
data CN   = Girl   | Boy   | Princess | Dwarf | Giant
          | Wizard | Sword | Dagger
          deriving Show
data RCN  = RCN1 CN That VP | RCN2 CN That NP TV
          deriving Show
data That = That deriving Show
data VP   = Laughed | Cheered | Shuddered
          | VP1 TV NP | VP2 DV NP NP
          deriving Show
data TV   = Loved    | Admired | Helped
          | Defeated | Caught
          deriving Show
data DV   = Gave deriving Show
```

Implementing Semantic Interpretation

- We define an interpretation function for every syntactic category
  - `SyntacticCategory -> SomeType`

Implementing Semantic Interpretation

- Expressions denoting relations are easiest: they are interpreted directly as relations in the model

```
intVP :: VP -> Entity -> Bool
intVP Laughed   = \ x -> laugh x
intVP Cheered   = \ x -> cheer x
intVP Shuddered = \ x -> shudder x

intTV :: TV -> Entity -> Entity -> Bool
intTV Loved    = \ x y -> love x y
intTV Admired  = \ x y -> admire x y
intTV Helped   = \ x y -> help x y
intTV Defeated = \ x y -> defeat x y

intDV :: DV -> Entity -> Entity -> Entity -> Bool
intDV Gave = \ x y z -> give x y z
```

Implementing Semantic Interpretation

- Expressions denoting relations are easiest: they are interpreted directly as relations in the model

```
intCN :: CN -> Entity -> Bool
intCN Girl     = \ x -> girl x
intCN Boy      = \ x -> boy x
intCN Princess = \ x -> princess x
intCN Dwarf    = \ x -> dwarf x
intCN Giant    = \ x -> giant x
intCN Wizard   = \ x -> wizard x
intCN Sword    = \ x -> sword x
intCN Dagger   = \ x -> dagger x
```

Implementing Semantic Interpretation

- Expressions denoting relations are easiest: they are interpreted directly as relations in the model

- N.B.: Using eta reduction, we could also have written

  ```
  intVP Laughed = laugh
  intTV Loved   = love
  intDV Gave    = give
  intCN Girl    = girl
  ```

  etc.

Implementing Semantic Interpretation

- Expressions denoting relations are easiest: they are interpreted directly as relations in the model

- N.B.: Using eta reduction, we could also have written

  ```
  intVP Laughed = laugh
  intTV Loved   = love
  intDV Gave    = give
  intCN Girl    = girl
  ```

  etc.

- Next: interpretation of determiners (quantifiers)

Computational Semantics
Day 3: Lambda calculus
and the composition of meanings

Jan van Eijck[1] & Christina Unger[2]

[1]CWI, Amsterdam, and UiL-OTS, Utrecht, The Netherlands
[2]CITEC, Bielefeld University, Germany

ESSLLI 2011, Ljubljana

Observation

Quantificational NPs do not refer to particular individuals.

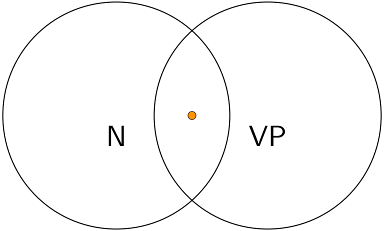
- *Every zombie bites someone.*
- *Nobody has seen a unicorn, because there aren't any!*

Maybe quantifiers indicate the quantity of something (all zombies, the empty set, and so on). But that's not exactly right, as it's not quantities that get predicated over (it's not the empty set that has seen a unicorn).

Rather, quantifiers relate sets.

Examples

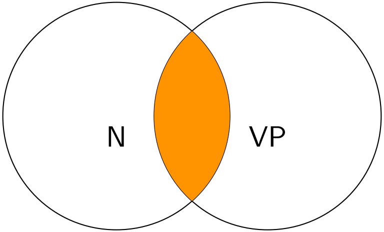
$[_{NP}$ *Some* $[_N$ *robot*$]]$ $[_{VP}$ *failed the Turing Test*$]$.



$$N \cap VP \neq \emptyset$$

Examples

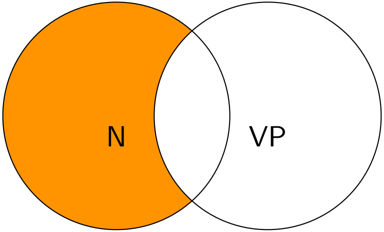
$[_{NP}$ *Every* $[_{N}$ *robot*$]]$ $[_{VP}$ *failed the Turing Test*$]$.



$$N - VP = \emptyset$$

Examples

$[_{NP}$ *No* $[_N$ *robot*$]]$ $[_{VP}$ *failed the Turing Test*$]$.



$$N \cap VP = \emptyset$$

Quantifiers as second-order predicates

Quantifiers can be expressed as second-order predicates of type $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$.

$$\begin{aligned}
[\![some]\!] &= \lambda P\, \lambda Q.\, \exists x.(P\ x) \wedge (Q\ x) \\
[\![every]\!] &= \lambda P\, \lambda Q.\, \forall x.(P\ x) \rightarrow (Q\ x) \\
[\![no]\!] &= \lambda P\, \lambda Q.\, \forall x.(P\ x) \rightarrow \neg(Q\ x) \\
&\quad \lambda P\, \lambda Q.\, \neg\exists x.(P\ x) \wedge (Q\ x)
\end{aligned}$$

Implementing Semantic Interpretation

- Interpretation of determiners as quantifiers in Haskell

```
intDET :: DET ->
         (Entity -> Bool) -> (Entity -> Bool) -> Bool

intDET Some p q = any q (filter p entities)

intDET Every p q = all q (filter p entities)

intDET No p q = not (intDET Some p q)
```

Implementing Semantic Interpretation

- Interpretation of determiners as quantifiers in Haskell

```
intDET The p q = singleton plist && q (head plist)
          where
              plist = filter p entities
              singleton [x] = True
              singleton  _  = False

intDET Most p q = length pqlist > length (plist \\ qlist)
    where
         plist  = filter p entities
         qlist  = filter q entities
         pqlist = filter q plist
```

Implementing Semantic Interpretation

- Determiner meanings are applied to common noun meanings to give noun phrase meanings

```
intNP :: NP -> (Entity -> Bool) -> Bool
intNP (NP1 det cn)  = (intDET det) (intCN cn)
```

Implementing Semantic Interpretation

- Determiner meanings are applied to common noun meanings to give noun phrase meanings

```
intNP :: NP -> (Entity -> Bool) -> Bool
intNP (NP1 det cn)  = (intDET det) (intCN cn)
```

- Sentence meanings are noun phrase meanings applied to verb phrase meanings

```
intSent :: Sent -> Bool
intSent (Sent np vp) = (intNP np) (intVP vp)
```

Example (the easy case)