Computational Semantics: Lambda Calculus

Scott Farrar
CLMA, University of Washington
farrar@u.washington.edu

February 24, 2010
Today’s lecture

1 Semantic Analysis
   - Problems

2 One Solution: \( \lambda \)-Calculus
   - \( \lambda \)-calculus and FOL
   - \( \lambda \)-calculus and compositionality

3 The semantics of words based on syntactic category
Semantic analysis

**Definition**

Semantic analysis is the derivation of a semantic representation from a string of words (perhaps marked up with syntactic structure). In other words, map sentences of NL onto logical formulas.

Map *Jim loves Betty* to *love*(JIM, BETTY)

There are several competing approaches for doing this, as there are several competing standards for the right semantic representation (use of event vs. relations).
Compositionality

Definition

Recall the principle of **compositionality**: the meaning of a complex expression is a function of the meaning of its parts.

The assumption is that we should be able to assign each “part” a meaning, then build larger structures, guided by the syntax of the language.

The syntax of NL and the syntax of predicate logic are similar, but ultimately not one-to-one compatible: translation between the two is a non-trivial task.
Event structure

A sailboat heels.
Event structure

\[
A \text{ sailboat heels.} \\
\exists e \exists b \ [\text{SailBoat}(b) \land \text{HeelingEvent}(e) \land \text{actor}(e, b)]
\]
Event structure

A sailboat heels.
\[ \exists e \exists b \ [ \text{SailBoat}(b) \land \text{HeelingEvent}(e) \land \text{actor}(e, b) ] \]

My sailboat is on the bottom.

\[ \exists e \ [ \text{SpatialLocating}(e) \land \text{theme}(e, \text{MYSB}) \land \text{loc}(e, \text{SEAFLOOR}) ] \]
Event structure

A sailboat heels.
\[ \exists e \ \exists b \ [\text{SailBoat}(b) \land \text{HeelingEvent}(e) \land \text{actor}(e, b)] \]

My sailboat is on the bottom.
\[ \exists e \ [\text{SpatialLocating}(e) \land \text{theme}(e, \text{MYSB}) \land \text{loc}(e, \text{SEAFLOOR})] \]
Semantic attachments

Consider the problem of two-place predicates in a non-event-style semantics: we need to map *Jim loves Betty* to something like:

\[ \text{love}(\text{JIM}, \text{BETTY}) \]
Semantic attachments

Consider the problem of two-place predicates in a non-event-style semantics: we need to map *Jim loves Betty* to something like:

\[ \text{love}(JIM, BETTY) \]

Let’s assume strict compositionality and say that the meaning of each syntactic constituent contributes to the meaning of the parent constituent. We could come up with something like \texttt{XP.sem} to stand for the semantics of some constituent XP.
Semantic attachments

Consider the problem of two-place predicates in a non-event-style semantics: we need to map Jim loves Betty to something like:

\[ love(JIM, BETTY) \]

Let’s assume strict compositionality and say that the meaning of each syntactic constituent contributes to the meaning of the parent constituent. We could come up with something like \( XP.sem \) to stand for the semantics of some constituent \( XP \).

Definition

**Semantic attachment** refers to the adornment of phrase structure rules with such semantic information.
Semantic attachments

```
S
 /   
NP   VP
 |   / 
NNP VBZ NP
  |  |   |
Jim loves NNP
   |     |
Betty

S-loves(JIM, BETTY)
```

```
NP-JIM
/     
NNP-JIM
|     |
Jim

VP-loves(x, BETTY)
/       
|       |
VBZ-loves(x, y)
|       |
loves

NP-BETTY
|   |
NNP-BETTY
|   |
Betty
```
Example: Semantic attachments

Assume that the + symbol stands for the compositionality operator:

\[ \text{loves}(JIM, BETTY) \]
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

$$S.\text{sem} = \text{NP.}\text{sem} + \text{VP.}\text{sem}$$

$$loves(JIM, BETTY)$$
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S\text{.sem} = \text{NP}\text{.sem} + \text{VP}\text{.sem}$
- $\text{VP}\text{.sem} = \text{V}\text{.sem} + \text{NP}\text{.sem}$

$$\text{loves}(\text{JIM}, \text{BETTY})$$
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S.sem = NP.sem + VP.sem$
- $VP.sem = V.sem + NP.sem$
- $V.sem = love.sem$

\[ \text{loves}(JIM, BETTY) \]
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S.\text{sem} = \text{NP.sem} + \text{VP.sem}$
- $\text{VP.sem} = \text{V.sem} + \text{NP.sem}$
- $\text{V.sem} = \text{love.sem}$
- $\text{love.sem} = \text{love}(x, y)$

$loves(\text{JIM}, \text{BETTY})$
Example: Semantic attachments

Assume that the + symbol stands for the compositionality operator:

- $S.\text{sem} = NP.\text{sem} + VP.\text{sem}$
- $VP.\text{sem} = V.\text{sem} + NP.\text{sem}$
- $V.\text{sem} = \text{love}.\text{sem}$
- $\text{love}.\text{sem} = love(x, y)$
- $NP.\text{sem} = NNP.\text{sem}$

\[
\text{loves}(\text{JIM}, \text{BETTY})
\]
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S.\text{sem} = \text{NP.}\text{sem} + \text{VP.}\text{sem}$
- $\text{VP.}\text{sem} = \text{V.}\text{sem} + \text{NP.}\text{sem}$
- $\text{V.}\text{sem} = \text{love.}\text{sem}$
- $\text{love.}\text{sem} = love(x, y)$
- $\text{NP.}\text{sem} = \text{NNP.}\text{sem}$
- $\text{NNP.}\text{sem} = \text{Betty.}\text{sem}$ or $\text{Jim.}\text{sem}$

\[\text{loves}(JIM, BETTY)\]
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S.sem = NP.sem + VP.sem$
- $VP.sem = V.sem + NP.sem$
- $V.sem = love.sem$
- $love.sem = love(x, y)$
- $NP.sem = NNP.sem$
- $NNP.sem = Betty.sem$ or $Jim.sem$
- $Betty.sem = BETTY$

$loves(JIM, BETTY)$
Example: Semantic attachments

Assume that the $+$ symbol stands for the compositionality operator:

- $S.sem = NP.sem + VP.sem$
- $VP.sem = V.sem + NP.sem$
- $V.sem = love.sem$
- $love.sem = love(x, y)$
- $NP.sem = NNP.sem$
- $NNP.sem = Betty.sem$ or $Jim.sem$
- $Betty.sem = BETTY$
- $Jim.sem = JIM$

$$loves(JIM, BETTY)$$
But what about other examples:
Analysis problem

But what about other examples:

- *Betty is loved by Jim.*
Analysis problem

But what about other examples:

- *Betty is loved by Jim.*
- *It’s Jim who loves Betty.*
But what about other examples:

- *Betty is loved by Jim.*
- *It’s Jim who loves Betty.*
- *Betty is the one loved by Jim.*
Analysis problem

But what about other examples:

- Betty is loved by Jim.
- It’s Jim who loves Betty.
- Betty is the one loved by Jim.

All clues to how the semantic representation might look are found in the syntactic structure of NL. All this, without even considering the ambiguity problem.
Analysis problem

The analysis problem: there is no (elegant) way to fill in the arguments of formulas at the level of semantic representation, in a way that is consistent with the syntax. In other words, there is no formal means of combining parts into wholes in standard FOL.

Even with passive verbs for example, we need to get BETTY to fill the second argument position of the predicate love(x, y).
Representation problem

Representation problem: no way to represent the meaning for some kinds of constituents.

We can very easily express the meaning of full sentences in plain FOL. We can say that a sentence is true given some state of the world.

John kissed Mary is true just in case John really did kiss Mary. With standard truth-conditional semantics, where the truth of propositions can either be $T$ or $F$, such logical expressions have a truth value. BUT...

The semantics of words based on syntactic category
Representation problem

Representation problem: no way to represent the meaning for some kinds of constituents.

We can very easily express the meaning of full sentences in plain FOL. We can say that a sentence is true given some state of the world. 

*John kissed Mary* is *T* just in case John really did kiss Mary.
Representation problem

Representation problem: no way to represent the meaning for some kinds of constituents.

We can very easily express the meaning of full sentences in plain FOL. We can say that a sentence is true given some state of the world. *John kissed Mary* is *T* just in case John really did kiss Mary.

With standard **truth-conditional semantics**, where the truth of propositions can either be *T* or *F*, such logical expressions have a truth value. BUT...
Representation problem

What about constituents like VPs: *kissed Opera*. The semantics would something like VP.sem, or $kiss(x, \text{OPRA})$.
Representation problem

What about constituents like VPs: *kissed* *Opra*. The semantics would something like VP.sem, or $\text{kiss}(x, \text{OPRA})$.
Representation problem

What about constituents like VPs: *kissed* *Opra*. The semantics would something like VP.sem, or *kiss(x, OPRA)*

But *kiss(x, OPRA)* has no truth value. This is because there are unbound variables: *x* has no connection to the *UD*. Such **open sentences** are neither *T* or *F*.
Representation problem

What about constituents like VPs: *kissed Opera*. The semantics would something like VP.sem, or \( \text{kiss}(x, \text{OPRA}) \)

- But \( \text{kiss}(x, \text{OPRA}) \) has no truth value. This is because there are unbound variables: \( x \) has no connection to the UD. Such open sentences are neither \( T \) or \( F \).
- Intuitively however, we know what a NL predicate/VP means: e.g., ... *kissed Opera* means something like a “kissing Opera event”, regardess of who does the kissing.
Representation problem

What about constituents like VPs: *kissed Opera*. The semantics would something like VP.sem, or $\textit{kiss}(x, \text{OPRA})$

- But $\textit{kiss}(x, \text{OPRA})$ has no truth value. This is because there are unbound variables: $x$ has no connection to the UD. Such open sentences are neither $T$ or $F$.
- Intuitively however, we know what a NL predicate/VP means: e.g., ... *kissed Opera* means something like a “kissing Opera event”, regardless of who does the kissing.
- But we cannot express the meaning of this in FOL given our current machinery, since we’ll always have an unbound variable.
Summary

In summary then, we have at least two problems for compositionality:

1. Analysis problem: No systematic way to use syntax to guide the construction of a semantic representation.
2. Representation problem: Unsatisfying approach to representing the meanings of certain constituents; deriving truth values for certain kinds of constituents is ill defined.
Today’s lecture

1. Semantic Analysis
   - Problems

2. One Solution: $\lambda$-Calculus
   - $\lambda$-calculus and FOL
   - $\lambda$-calculus and compositionality

3. The semantics of words based on syntactic category
Alonzo Church created a calculus for describing arbitrary functions, called \( \lambda \text{-calculus} \). (It was developed to give a functional foundation for mathematics.) It wasn’t picked up by mathematicians, but it did become a versatile tool for computer scientists.
Alonzo Church created a calculus for describing arbitrary functions, called \( \lambda \)-calculus. (It was developed to give a functional foundation for mathematics.) It wasn’t picked up by mathematicians, but it did become a versatile tool for computer scientists.

Remember Lisp? The second oldest high-level programming language, and still used today (invented by John McCarthy, 1958). Lisp (pure Lisp at least) deals exclusively with functions, and functions can be created on the fly and without names.
back to Church

Alonzo Church created a calculus for describing arbitrary functions, called \( \lambda\)-calculus. (It was developed to give a functional foundation for mathematics.) It wasn’t picked up by mathematicians, but it did become a versatile tool for computer scientists.

Remember **Lisp**? The second oldest high-level programming language, and still used today (invented by John McCarthy, 1958). Lisp (pure Lisp at least) deals exclusively with functions, and functions can be created on the fly and without names.

In Lisp, this expression evaluates to an anonymous function: 
\[
(\text{\texttt{lambda (x y) (+ x y)})}, \text{ read as “the pair x and y are mapped to x + y”}
\]
Alonzo Church created a calculus for describing arbitrary functions, called $\lambda$-calculus. (It was developed to give a functional foundation for mathematics.) It wasn’t picked up by mathematicians, but it did become a versatile tool for computer scientists.

Remember **Lisp**? The second oldest high-level programming language, and still used today (invented by John McCarthy, 1958). Lisp (pure Lisp at least) deals exclusively with functions, and functions can be created on the fly and without names.

In Lisp, this expression evaluates to an anonymous function: \( \text{(lambda (x y) (+ x y))} \), read as “the pair \( x \) and \( y \) are mapped to \( x + y \)”. 

Alonzo Church created a calculus for describing arbitrary functions, called \( \lambda \)-calculus. (It was developed to give a functional foundation for mathematics.) It wasn’t picked up by mathematicians, but it did become a versatile tool for computer scientists.

Remember **Lisp**? The second oldest high-level programming language, and still used today (invented by John McCarthy, 1958). Lisp (pure Lisp at least) deals exclusively with functions, and functions can be created on the fly and without names.

In Lisp, this expression evaluates to an anonymous function: 

\[
(\text{lambda} \ (x \ y) \ (+ \ x \ y))
\]

read as “the pair \( x \) and \( y \) are mapped to \( x + y \)”. Otherwise, we’d have a named function, something like: 

\[
\text{add}(x, y)
\]
More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: \((+ x y)\):

Start with three symbols: +, x, and y. Treat each symbol as either a function or argument. 

- \(+ x\) yields \((+ x)\)
- \((+ x) y\) yields \(((+ x) y)\)

Thus, when an expression (function) is applied to another expression (argument), a third expression (result) is obtained.
More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: (+ x y):

- Start with three symbols: +, x, and y
More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: $(+ \times y)$:

- Start with three symbols: $+$, $x$, and $y$
- Treat each symbol as either a function or argument
More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: (+ x y):

- Start with three symbols: +, x, and y
- Treat each symbol as either a function or argument
- + x yields (+ x)

Thus, when an expression (function) is applied to another expression (argument), a third expression (result) is obtained.
More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: \((+ \times y)\):

- Start with three symbols: \(+\), \(\times\), and \(y\)
- Treat each symbol as either a function or argument
- \(+ \times y\) yields \((+ \times)\)
- \((+ \times) y\) yields \(((+\times)y)\)
Functions and arguments

More generally, we can describe what’s going on by assuming that every expression is either a function or argument. For instance, suppose we want to create: $(+ x y)$:

- Start with three symbols: $+$, $x$, and $y$
- Treat each symbol as either a function or argument
- $+ x$ yields $(+ x)$
- $(+ x) y$ yields $((+x)y)$

Thus, when an expression (function) is applied to another expression (argument), a third expression (result) is obtained.
λ-calculus: Formal definition

**Definition**

Expressions in the language $\Lambda$ are composed of:

- variables $\{a, b, c, \ldots, x, y, z\}$
- abstraction symbols: $\lambda$ and $.$, the dot
- parentheses: ( and )

**λ-terms.** $T \in \Lambda$ iff one of the following holds:

1. $T$ is a member of a countable set of variables
2. $T$ is of the form $(MN)$ where $M$ and $N$ are in $\Lambda$.
3. $T$ is of the form $(\lambda X.Y)$ where $X$ is a variable and $Y$ is in $\Lambda$.

$\Lambda$ is the smallest language with this property.

$(MN)$ is called an **application** and $\lambda X.Y$ is an **abstraction**.
Examples

The following are all examples of \( \lambda \) expressions:

1. \( \lambda x. x \)
2. \( \lambda x. y (\lambda x. z x) \)
3. \( \lambda x. x (y) \)
Examples

The following are all examples of $\lambda$ expressions:

1. $\lambda x . x$
Examples

The following are all examples of \( \lambda \) expressions:

1. \( \lambda x . x \)
2. \( \lambda x . y (\lambda x . z x) \)
Examples

The following are all examples of $\lambda$ expressions:

1. $\lambda x . x$
2. $\lambda x . y (\lambda x . z x)$
3. $\lambda x . x (y)$
Examples

The following are all examples of λ expressions:

1. \( \lambda x . x \)
2. \( \lambda x . y (\lambda x . z x) \)
3. \( \lambda x . x (y) \)
Standard definitions of FOL can be augmented with λ-calculus. The point is that we can use standard FOL formulas as functions and create new FOL formulas compositionally.
λ-calculus and FOL

Standard definitions of FOL can be augmented with λ-calculus. The point is that we can use standard FOL formulas as functions and create new FOL formulas compositionally.

Definition

If in some formula a variable is bound by the λ operator, the formula is called a lambda expression.
\(\lambda\)-calculus and FOL

Standard definitions of FOL can be augmented with \(\lambda\)-calculus. The point is that we can use standard FOL formulas as functions and create new FOL formulas compositionally.

Definition

If in some formula a variable is bound by the \(\lambda\) operator, the formula is called a \textbf{lambda expression}.

Syntactically, a \(\lambda\)-expression looks just like any other quantified expression:

\[
\lambda x.\text{red}(x)
\]

\[
\forall y.\text{boat}(y)
\]

\[
\exists z.\text{floats}(z)
\]
Application expressions

To symbolize compositionality, we can create a new formula from $\lambda x.\text{dog}(x)$ by treating it as a **function** and then applying it to an **argument**:

$$\lambda x.\text{dog}(x)(\text{FIDO})$$
Application expressions

To symbolize compositionality, we can create a new formula from $\lambda x.\text{dog}(x)$ by treating it as a function and then applying it to an argument:

$$\lambda x.\text{dog}(x)(\text{FIDO})$$
Application expressions

To symbolize compositionality, we can create a new formula from \( \lambda x.\text{dog}(x) \) by treating it as a **function** and then applying it to an **argument**:

\[
\lambda x.\text{dog}(x)(FIDO)
\]

The result is:

\[
\text{dog}(FIDO)
\]
Application expressions

To symbolize compositionality, we can create a new formula from $\lambda x.\text{dog}(x)$ by treating it as a **function** and then applying it to an **argument**:

$$\lambda x.\text{dog}(x)(FIDO)$$

The result is:

$$\text{dog}(FIDO)$$

**Definition**

Given some application expression $F A$ the function can be reduced by a process called **$\beta$-reduction**, such that the result is $F$ with all occurrences of variables bound by $\lambda$ replaced by $A$. (The terminology has roots in the original papers of Church and Kleene.)
The λ operator is represented by the single back slash \, and is indicated with a raw string:

\ x . dog(x) (FIDO)

The Python string is the equivalent of the following application expression:
\( \lambda x.\text{dog} \ x \ (FIDO) \)
Scope of $\lambda$

In the augmented FOL, the $\lambda$ operator ranges over sets and individuals, not just individuals as with $\forall$ and $\exists$.

**example**

$$ (\lambda P. P) \ (\text{walk}(x)) $$

reduces to:

$$ \text{boat}(x) $$
Summary of terminology

- **abstraction**: the process of creating a $\lambda$ function from a predicate logic formula.
Summary of terminology

- **abstraction**: the process of creating a λ function from a predicate logic formula.
- **λ expression**: one with variables bound by the λ operator, sometimes called a λ function.
Summary of terminology

- **abstraction**: the process of creating a $\lambda$ function from a predicate logic formula.
- **$\lambda$ expression**: one with variables bound by the $\lambda$ operator, sometimes called a $\lambda$ function.
- **application expression**: one with a function and an argument.
Summary of terminology

- **abstraction**: the process of creating a $\lambda$ function from a predicate logic formula.
- **$\lambda$ expression**: one with variables bound by the $\lambda$ operator, sometimes called a $\lambda$ function.
- **application expression**: one with a function and an argument.
- **$\beta$-reduction**: where subparts of a function are evaluated and rewritten until the function itself is reduced to a simpler form.
Steps in compositionality deriving a semantic representation:

1. Express the semantics of each constituent in terms of lambda expressions;
2. Determine which expression is the function and which is the argument;
3. Apply the function to the argument;
4. β-reduce the conjoined elements to arrive at the final semantic representation.
Steps in compositionality

Steps in compositionally deriving a semantic representation:

1. Express the semantics of each constituent in terms of lambda expressions;
Steps in compositionality deriving a semantic representation:

1. Express the semantics of each constituent in terms of lambda expressions;
2. Determine which expression is the function and which is the argument;
Steps in compositionality

Steps in compositionally deriving a semantic representation:

1. Express the semantics of each constituent in terms of lambda expressions;
2. Determine which expression is the function and which is the argument;
3. Apply the function to the argument;
Steps in compositionality

Steps in compositionally deriving a semantic representation:

1. Express the semantics of each constituent in terms of lambda expressions;
2. Determine which expression is the function and which is the argument;
3. Apply the function to the argument;
4. $\beta$-reduce the conjoined elements to arrive at the final semantic representation.
Example: *Sue bikes*

\[ Sue \text{ bikes } \Rightarrow \text{bikes}(\text{SUE}) \]

Given:
Example: \textit{Sue bikes}

\[ \textit{Sue bikes} \Rightarrow \textit{bikes}(SUE) \]

Given:

- \( \lambda \)-expression for \textit{Sue}: \( \lambda P \cdot P \ (\text{SUE}) \)
Example: *Sue bikes*

$Sue \ bikes \Rightarrow bikes(SUE)$

Given:

- $\lambda$-expression for *Sue*: $\ \lambda\ P \ . \ P\ (SUE)$
- $\lambda$-expression for *bikes*: $\ \lambda\ x. \ \text{bikes}(x)$
Example: *Sue bikes*

\[ \text{Sue bikes} \Rightarrow \text{bikes}(SUE) \]

**Given:**

- λ-expression for *Sue*: \( \lambda P \cdot P(SUE) \)
- λ-expression for *bikes*: \( \lambda x \cdot \text{bikes}(x) \)

**Derivation**

An application expression:

\[ \lambda P \cdot P(SUE) (\lambda x \cdot \text{bikes}(x)) \]

\[ \text{bikes}(SUE) \text{ by } \beta \text{-reduction} \]

25/37
Example: *Sue bikes*

\[ Sue \text{ bikes} \Rightarrow \text{bikes}(SUE) \]

Given:

- \( \lambda \)-expression for *Sue*: \( \lambda P . P (SUE) \)
- \( \lambda \)-expression for *bikes*: \( \lambda x . \text{bikes}(x) \)

Derivation

- An application expression:
  \[ \lambda P . P (SUE) (\lambda x . \text{bikes}(x)) \]
Example: *Sue bikes*

\[ \text{Sue bikes } \Rightarrow \text{bikes}(\text{SUE}) \]

**Given:**

- \(\lambda\)-expression for *Sue*: \(\lambda P. P(\text{SUE})\)
- \(\lambda\)-expression for *bikes*: \(\lambda x. \text{bikes}(x)\)

**Derivation**

- An application expression:
  \[ \lambda P. P(\text{SUE})(\lambda x. \text{bikes}(x)) \]
- \(\lambda x. \text{bikes}(x)(\text{SUE})\) by \(\beta\)-reduction
Example: *Sue bikes*

\[
Sue \text{ bikes } \Rightarrow \text{ bikes}(SUE)
\]

Given:

- $\lambda$-expression for *Sue*: $\lambda P \ P \ (SUE)$
- $\lambda$-expression for *bikes*: $\lambda x \ \text{bikes}(x)$

**Derivation**

- An application expression:
  \[
  \lambda P \ P \ (SUE) \ (\lambda x \ \text{bikes}(x))
  \]
- $\lambda x \ \text{bikes}(x) \ (SUE)$ by $\beta$-reduction
- bikes(SUE) by $\beta$-reduction
Today’s lecture

1 Semantic Analysis
   • Problems

2 One Solution: λ-Calculus
   • λ-calculus and FOL
   • λ-calculus and compositionality

3 The semantics of words based on syntactic category
General strategy for using \( \lambda \)-calculus

The point is to enrich each lexical entry with a semantics, and then derive the semantic representation of the entire sentence or phrase.

We’ll need to express the semantics of everything using \( \lambda \)-calculus. Namely, we’ll need to express the semantics of lexical items using the functional notation. \( NNP \rightarrow Sue \)
\( NNP[sem = \lambda \ S . S (SUE)] \rightarrow Sue \)
Intransitive verbs

Intransitive verbs, in non-event style FOL, are mapped to unary predicates. The semantic attachment for run would be $\lambda x.\text{run}(x)$, a predicate waiting for an argument. *Bill runs:*
Intransitive verbs, in non-event style FOL, are mapped to unary predicates. The semantic attachment for run would be $\lambda x.\text{run}(x)$, a predicate waiting for an argument. Bill runs:

\[
\lambda x.\text{run}(x) \ (\text{BILL}) \text{ reduces to: } \text{run}(\text{BILL})
\]
Intransitive verbs, in non-event style FOL, are mapped to unary predicates. The semantic attachment for *run* would be $\lambda x.\text{run}(x)$, a predicate waiting for an argument. *Bill runs:*

\[
\lambda x.\text{run}(x) \text{ (BILL)} \text{ reduces to: } \text{run}(\text{BILL})
\]

But, Bill comes before the verb in the syntax.
Proper nouns

Ordinarily, the semantic attachment for *Bill* would be a constant like *BILL*, as proper nouns are (non-logical) constants, i.e., always arguments of other expressions. But in a λ system, the semantic attachment is \( \lambda \cdot \lambda (\text{BILL}) \). Why? Because we need the semantics of a proper noun to be a function in order to get our representations to come out correctly.
Ordinarily, the semantic attachment for *Bill* would be a constant like `BILL`, as proper nouns are (non-logical) constants, i.e., always arguments of other expressions. But in a λ system, the semantic attachment is \( P \ . \ P \ (\text{BILL}) \).

Why? Because we need the semantics of a proper noun to be a function in order to get our representations to come out correctly.
We need to preserve the order from the syntax. For *Bill runs*, we need to find a semantic representation for the word *Bill* and then for *runs*:
Intransitive verbs, proper order

We need to preserve the order from the syntax. For *Bill runs*, we need to find a semantic representation for the word *Bill* and then for *runs*:

\[
(\lambda P. P) (\text{BILL}) (\lambda x. \text{run}(x)) \text{ reduces to: }
\lambda x. \text{run}(x) (\text{BILL})
\]

\text{run}(\text{BILL})
Intransitive verbs, proper order

We need to preserve the order from the syntax. For *Bill runs*, we need to find a semantic representation for the word *Bill* and then for *runs*:

\[ (\lambda P. P) (\text{BILL}) (\lambda x. \text{run}(x)) \]

reduces to:

\[ \lambda x. \text{run}(x) \text{(BILL)} \]

\[ \text{run(BILL)} \]

Thus, order from the syntax can be used as is, which makes things much easier for compositionality.
More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:
Transitive Verbs

More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:

\[ \lambda y. \lambda x. \text{love}(x,y) \]

\[ \text{love}(JIM, BETTY) \]
Transitive Verbs

More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:

\[
\begin{align*}
\lambda y. \lambda x. & \text{love}(x,y) \\
\lambda y. \lambda x. & \text{love}(x,y) \text{ (BETTY)}
\end{align*}
\]
More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:

1. $\lambda y. \lambda x. \text{love}(x,y)$
2. $\lambda y. \lambda x. \text{love}(x,y)$ (BETTY)
3. $\lambda x. \text{love}(x,\text{BETTY})$
Transitive Verbs

More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:

- \( \lambda y. \lambda x. \text{love}(x, y) \)
- \( \lambda y. \lambda x. \text{love}(x, y) \) (BETTY)
- \( \lambda x. \text{love}(x, \text{BETTY}) \)
- \( \lambda x. \text{love}(x, \text{BETTY}) \) (JIM)
Transitive Verbs

More care has to be taken to specify the order of reduction for the semantics of transitive verbs and di-transitive verbs. These are respectively binary and ternary predicates in FOL. For a transitive verb like love:

- \( \lambda y. \lambda x. \text{love}(x,y) \)
- \( \lambda y. \lambda x. \text{love}(x,y) \) (BETTY)
- \( \lambda x. \text{love}(x,\text{BETTY}) \)
- \( \lambda x. \text{love}(x,\text{BETTY}) \) (JIM)
- love(JIM,\text{BETTY})
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since \textit{JIM} is not a function:
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since \textit{JIM} is not a function:

Consider:
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since JIM is not a function:

Consider:

\ Q. Q (JIM) which is another form of simply JIM
But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since JIM is not a function:

Consider:

- \( \lambda Q. Q (\text{JIM}) \) which is another form of simply JIM
- \( \lambda X. \lambda y. X(\lambda x. \text{loves}(y,x)) \)
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since JIM is not a function:

Consider:

- \( Q \ Q \ (JIM) \) which is another form of simply JIM
- \( X \ y. \ X(\ x. \ loves(y,x)) \)
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since *JIM* is not a function:

Consider:

- \ Q. Q (JIM) \ which \ is \ another \ form \ of \ simply \ JIM
- \ X \ y. \ X(\ x. \ loves(y,x))

- \ X \ y. \ X (\ x. \ loves(y,x)) ( \ Q . Q (BETTY))
  just the inner terms
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since JIM is not a function:

Consider:

- $\lambda Q. Q (\text{JIM})$ which is another form of simply JIM
- $\lambda X \ y. \ X(\lambda x. \ \text{loves}(y,x))$

- $\lambda X \ y. \ X (\lambda x. \ \text{loves}(y,x)) (\lambda Q . \ Q (\text{BETTY}))$
  
  just the inner terms

- $(\lambda Q . \ Q (\text{BETTY})) (\lambda x. \ \text{loves}(y,x))$
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since JIM is not a function:

Consider:

\[ Q \cdot Q \ (\mathrm{JIM}) \] which is another form of simply JIM

\[ X \ y.\ X(\ \ x. \ \ loves(y,x)) \]

\[ X \ y.\ X (\ \ x. \ \ loves(y,x)) (\ Q \cdot Q \ (\mathrm{BETTY})) \]

just the inner terms

\[ (\ Q \cdot Q \ (\mathrm{BETTY})) (\ x. \ \ loves(y,x)) \]

\[ x. \ \ loves(y,x) \ (\mathrm{BETTY}) \]
Transitive Verbs, proper order

But, how do we deal with the linear order of the NL string? Due to subject and object order, the following will not reduce, since \textit{JIM} is not a function:

Consider:

\[ \lambda \quad Q. \quad Q \text{(JIM)} \text{ which is another form of simply JIM} \]

\[ \lambda \quad X \quad y. \quad X(\lambda \quad x. \quad \text{loves}(y,x)) \]

\[ \lambda \quad X \quad y. \quad X \left( \lambda \quad x. \quad \text{loves}(y,x) \right) \left( \lambda \quad Q. \quad Q \text{(BETTY)} \right) \]

\[ \text{just the inner terms} \]

\[ \left( \lambda \quad Q. \quad Q \text{(BETTY)} \right) \left( \lambda \quad x. \quad \text{loves}(y,x) \right) \]

\[ \lambda \quad x. \quad \text{loves}(y,x) \text{(BETTY)} \]

\[ \text{loves}(y,BETTY) \]
Transitive Verbs, proper order

Semantic Analysis Problems
One Solution: \( \lambda \)-Calculus
\( \lambda \)-calculus and FOL
\( \lambda \)-calculus and compositionality

The semantics of words based on syntactic category
Transitive Verbs, proper order

\( \lambda \cdot \text{loves}(y, \text{BETTY}) \)
Transitive Verbs, proper order

\[ \lambda y. \text{loves}(y, \text{BETTY}) \]

\[ \lambda \text{P} . \text{P}(\text{JIM})(\lambda y. \text{loves}(y, \text{BETTY})) \]
Transitive Verbs, proper order

- \( y . \text{loves}(y, \text{BETTY}) \)
- \( P . P (\text{JIM}) (\lambda y . \text{loves}(y, \text{BETTY})) \)
- \( \lambda y . \text{loves}(y, \text{BETTY}) (\text{JIM}) \)
Transitive Verbs, proper order

- \( y \cdot \text{loves}(y, \text{BETTY}) \)
- \( P \cdot P(\text{JIM})(\lambda y. \text{loves}(y, \text{BETTY})) \)
- \( \lambda y. \text{loves}(y, \text{BETTY})(\text{JIM}) \)
- \( \text{loves}(\text{JIM}, \text{BETTY}) \)
Full transitive verb example

And the mostly unreadable full lambda expression for *Jim loves Betty*:

\[ P . P (JIM) (\ X \ y . \ X(\ x . \ loves(y,x)) ( \ Q . \ Q (BETTY))) \]
Common nouns work just like intransitive verbs, i.e., the semantic attachment is a unary predicate.
Nouns

Common nouns work just like intransitive verbs, i.e., the semantic attachment is a unary predicate.

For example, the semantic attachment for *dog* would be:
\[
\lambda x.\text{dog}(x) \text{ in } \lambda\text{-calculus.}
\]
Copulas

The copula (am, is, are, etc.) is a special kind of transitive verb, as it equates the subject and object. We introduce a special binary predicate $eq$ for the semantics of the copula:

$$\lambda x y. X(\lambda x. eq(y, x))$$
Copulas

The copula (am, is, are, etc.) is a special kind of transitive verb, as it equates the subject and object. We introduce a special binary predicate eq for the semantics of the copula:

\[ \lambda X \ y. \ X(\lambda x. \ eq(y,x)) \]

The semantics of the copula looks just like the semantics of any transitive verb (see previous).
The copula (am, is, are, etc.) is a special kind of transitive verb, as it equates the subject and object. We introduce a special binary predicate $eq$ for the semantics of the copula:
\[
\lambda X \, \lambda y. \ X(\lambda x. \ eq(y,x))
\]

The semantics of the copula looks just like the semantics of any transitive verb (see previous).

For the negative copula (ain’t, isn’t, etc.) we have a slightly different formula:
\[
\lambda X \, \lambda y. \ X(\lambda x. \ -eq(y,x))
\]
Auxiliaries

An auxiliary verb such as *does* is transparent at the level of semantic representation, at least concerning propositional content. Thus, *does go* would simply be:

\[ \lambda z. \text{go}(z). \]
Auxiliaries

An auxiliary verb such as *does* is transparent at the level of semantic representation, at least concerning propositional content. Thus, *does go* would simply be:

\[ \lambda z. \text{go}(z). \]

If we want to specify a semantics for *does* that will turn out to contribute nothing to higher constituents, this will suffice: The lambda expression for the semantics of an auxiliary contributing nothing would be:

\[ \lambda P \ x. \ P(x) (\lambda z. \text{go}(z)) \]
Auxiliaries

An auxiliary verb such as *does* is transparent at the level of semantic representation, at least concerning propositional content. Thus, *does go* would simply be:

\[ \lambda z. \text{go}(z) \. \]

If we want to specify a semantics for *does* that will turn out to contribute nothing to higher constituents, this will suffice: The lambda expression for the semantics of an auxiliary contributing nothing would be:

\[ \lambda P \ x. \ P(x) \ (\lambda z. \text{go}(z)) \]

- *does go*
Auxiliaries

An auxiliary verb such as *does* is transparent at the level of semantic representation, at least concerning propositional content. Thus, *does go* would simply be:

\( \lambda z. \text{go}(z) \).

If we want to specify a semantics for *does* that will turn out to contribute nothing to higher constituents, this will suffice:

The lambda expression for the semantics of an auxiliary contributing nothing would be:

\( \lambda P \ x. \ P(x) (\lambda z. \text{go}(z)) \)

- *does go*
- \( \lambda P \ x. \ P(x) (\lambda z. \text{go}(z)) \)
Auxiliaries

An auxiliary verb such as does is transparent at the level of semantic representation, at least concerning propositional content. Thus, does go would simply be:
\[ \text{does go} = \lambda z. \text{go}(z). \]

If we want to specify a semantics for does that will turn out to contribute nothing to higher constituents, this will suffice:

The lambda expression for the semantics of an auxiliary contributing nothing would be:
\[ \lambda P. \lambda x. P(x) (\lambda z. \text{go}(z)) \]

- does go
- \[ \lambda P. \lambda x. P(x) (\lambda z. \text{go}(z)) \]
- \[ \lambda x. (\lambda z. \text{go}(z)) (x) \]
Auxiliaries

An auxiliary verb such as *does* is transparent at the level of semantic representation, at least concerning propositional content. Thus, *does go* would simply be:
\[ \lambda z. \text{go}(z). \]

If we want to specify a semantics for *does* that will turn out to contribute nothing to higher constituents, this will suffice: The lambda expression for the semantics of an auxiliary contributing nothing would be:
\[ \lambda P \ x. \ P(x) (\lambda z. \text{go}(z)) \]

*does go*

\[ \lambda P \ x. \ P(x) (\lambda z. \text{go}(z)) \]
\[ \lambda x . (\lambda z. \text{go}(z))(x) \]
\[ \lambda z . \text{go}(z) \text{ (same as we started with; *does* is transparent)} \]