# Computational Semantics: More $\lambda$ Calculus

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

March 1, 2010

## Today's lecture

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.
- It is a calculus of functions and function application $(F\ A)$, where $F$ is some function and $A$ is some argument.

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.
- It is a calculus of functions and function application $(F\ A)$, where $F$ is some function and $A$ is some argument.
- $F$ is in the form of $\lambda var.expr$ such that $var$ is bound by the $\lambda$ operator.

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.
- It is a calculus of functions and function application $(F\ A)$, where $F$ is some function and $A$ is some argument.
- $F$ is in the form of $\lambda var.expr$ such that $var$ is bound by the $\lambda$ operator.
- $\lambda x.red(x)$ is an example of a $\lambda$-expression.

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.
- It is a calculus of functions and function application $(F\ A)$, where $F$ is some function and $A$ is some argument.
- $F$ is in the form of $\lambda var.expr$ such that $var$ is bound by the $\lambda$ operator.
- $\lambda x.red(x)$ is an example of a $\lambda$-expression.
- The function $\lambda x.red(x)$ is **anonymous**; it has no name.

## Key points from last time

- The $\lambda$-calculus can be considered an axiomatic **theory of functions**.
- It is a calculus of functions and function application $(F\ A)$, where $F$ is some function and $A$ is some argument.
- $F$ is in the form of $\lambda var.expr$ such that $var$ is bound by the $\lambda$ operator.
- $\lambda x.red(x)$ is an example of a $\lambda$-expression.
- The function $\lambda x.red(x)$ is **anonymous**; it has no name.
- The $\lambda$-calculus can be used with FOL to functions to aid in the **compositionality** process.

## Today's lecture

1. $\lambda$-calculus Recap
2. NLTK semantics
3. $\lambda$ operations
4. Type theory

## NLTK semantics

### Variables

The NLTK implements FOL and $\lambda$-calculus starting with a basic functional calculus and **then** adding elements of FOL. Furthermore, variables in the NLTK's implementation are typed:

## NLTK semantics

### Variables

The NLTK implements FOL and $\lambda$-calculus starting with a basic functional calculus and **then** adding elements of FOL. Furthermore, variables in the NLTK's implementation are typed:

- `IndividualVariableExpression`: the value has to be `a, b, c, ..., w,x,y,z` (but not `e`), plus 0 or more numerals, e.g., `x, y, x1, y23`.

## NLTK semantics

### Variables

The NLTK implements FOL and $\lambda$-calculus starting with a basic functional calculus and **then** adding elements of FOL. Furthermore, variables in the NLTK's implementation are typed:

- `IndividualVariableExpression`: the value has to be `a, b, c, ..., w,x,y,z` (but not `e`), plus 0 or more numerals, e.g., `x, y, x1, y23`.
- `EventVariableExpression`: has to be `e` or `e1, e2, ...`

## NLTK semantics

### Variables

The NLTK implements FOL and $\lambda$-calculus starting with a basic functional calculus and **then** adding elements of FOL. Furthermore, variables in the NLTK's implementation are typed:

- `IndividualVariableExpression`: the value has to be `a, b, c, ..., w,x,y,z` (but not `e`), plus 0 or more numerals, e.g., `x, y, x1, y23`.
- `EventVariableExpression`: has to be `e` or `e1, e2, ...`
- `FunctionVariableExpression`: has to be a single capital letter and can be followed by a numeral, e.g., `A`, `B`, `A1`, `E1`

## NLTK semantics

### Constants

- `ConstantExpression`: an expression consisting of a constant, e.g., `BILL`, `BB`, `bill`

## NLTK semantics

### Binder expressions

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

# NLTK semantics

## Binder expressions

- `VariableBinderExpression`: an abstract class, an expression with at least one bound variable and a binding operator (`\`, `all`, `exists`)

## NLTK semantics

### Binder expressions

- `VariableBinderExpression`: an abstract class, an expression with at least one bound variable and a binding operator (\, `all`, `exists`)
- `LambdaExpression`: an expression with at least one variable bound by the $\lambda$ operator (\)

## NLTK semantics

### Binder expressions

- `VariableBinderExpression`: an abstract class, an expression with at least one bound variable and a binding operator (\, `all`, `exists`)
- `LambdaExpression`: an expression with at least one variable bound by the $\lambda$ operator (\)
- `ExistsExpression`: an expression with at least one variable bound by the `exists` operator

## NLTK semantics

### Binder expressions

- `VariableBinderExpression`: an abstract class, an expression with at least one bound variable and a binding operator (`\`, `all`, `exists`)
- `LambdaExpression`: an expression with at least one variable bound by the $\lambda$ operator (`\`)
- `ExistsExpression`: an expression with at least one variable bound by the `exists` operator
- `AllExpression`: an expression with at least one variable bound by the `all` operator

## NLTK semantics

### Binder expressions

- `VariableBinderExpression`: an abstract class, an expression with at least one bound variable and a binding operator (\, `all`, `exists`)
- `LambdaExpression`: an expression with at least one variable bound by the $\lambda$ operator (\)
- `ExistsExpression`: an expression with at least one variable bound by the `exists` operator
- `AllExpression`: an expression with at least one variable bound by the `all` operator
- `ApplicationExpression`: an expression with a functor and an argument

## Summary of $\lambda$-expressions

| **syn. category** | **example** | **FOL** | **$\lambda$ expression** |
|---|---|---|---|
| common noun | dog | $dog(x)$ | `\ x.dog(x)` |
| proper noun | Bill | $BILL$ | `\ P.P(BILL)` |
| intransitive verb | runs | $run(x)$ | `\ x.run(x)` |
| transitive verb | loves | $love(x, y)$ | `\ X y.X(\ x.love(y,x))` |
| copula | is | $eq(x, y)$ | `\ X y.X(\ x.eq(y,x))` |
| negative copula | isn't | $\neg eq(x, y)$ | `\ X y.X(\ x.-eq(y,x))` |
| auxiliary verb | did go | $go(x)$ | `\ K z.K(z) (\ x.go(x))` |
| neg. auxiliary verb | didn't go | $\neg go(x)$ | `\ K z.-K(z) (\ x.go(x))` |

## Abstractions

We say that:
$\backslash$ `x.red(x)` is a $\lambda$ abstraction

## Abstractions

We say that:
$\backslash$ `x.red(x)` is a $\lambda$ abstraction

**Definition**

The term $\lambda$-**abstraction** refers to a function, possibly constructed from an expression which was not originally a function, e.g., a predicate logic formula.

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

## Applications

We say that:

```
\ x.  red(x) (BOAT)
```

## Applications

We say that:

```
\ x.  red(x) (BOAT)
```

is an application expression.

## Applications

We say that:

```
\ x.  red(x) (BOAT)
```

is an application expression.

**Definition**

An **application expression** is a formula with a **function** and an **argument**.

## Today's lecture

## Reducing

We say that:
$\backslash x.red(x)(BOAT)$
$\beta$-reduces to:
`red(BOAT)`

**Definition**

$\beta$-**reduction** is the process of substituting an argument for variables (in the function) bound by the $\lambda$ operator.

## $\alpha$-conversion

**Definition**

**Alpha conversion** allows bound variable names to be changed. For example, an alpha conversion of `\ x.x` would be `\ y.y` . Frequently in uses of $\lambda$ calculus, terms that differ only by alpha conversion are considered to be equivalent.

$\backslash$ x.x $\equiv$ $\backslash$ y.y $\equiv$ $\backslash$ t.t

Given `\ x.  \ x.x`, which of the following would be a valid $\alpha$-conversion?

1. `\ y.  \ x.x`
2. `\ y.\ x.y`

Scott Farrar CLMA, University of Washington far-rar@u.washington.ed

## $\alpha$-conversion

**Definition**

**Alpha conversion** allows bound variable names to be changed. For example, an alpha conversion of \ x.x would be \ y.y . Frequently in uses of $\lambda$ calculus, terms that differ only by alpha conversion are considered to be equivalent.

\ x.x $\equiv$ \ y.y $\equiv$ \ t.t

Given \ x. \ x.x, which of the following would be a valid $\alpha$-conversion?

1. \ y. \ x.x
2. \ y.\ x.y (invalid conversion)

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

## Today's lecture

1. $\lambda$-calculus Recap
2. NLTK semantics
3. $\lambda$ operations
4. Type theory

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

For example, the target semantics for a noun modified by an adjective would be *red ball*, would translate to:

```
\ x.red(x) & ball(x)
```

The result is obtained using this lambda expression for *red*:

```
\ P y.  (red(y) & P(y))
```

Scott Farrar
CLMA, University of Washington far-rar@u.washington.ed

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

For example, the target semantics for a noun modified by an adjective would be *red ball*, would translate to:

```
\ x.red(x) & ball(x)
```

The result is obtained using this lambda expression for *red*:

```
\ P y.  (red(y) & P(y))
```

- *red ball*

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

For example, the target semantics for a noun modified by an adjective would be *red ball*, would translate to:
`\ x.red(x) & ball(x)`
The result is obtained using this lambda expression for *red*:
`\ P y.  (red(y) & P(y))`

- *red ball*
- `\ P y.  (red(y) & P(y)) (\ x.ball(x))`

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

For example, the target semantics for a noun modified by an adjective would be *red ball*, would translate to:
```
\ x.red(x) & ball(x)
```
The result is obtained using this lambda expression for *red*:
```
\ P y.  (red(y) & P(y))
```

- *red ball*
- `\ P y.  (red(y) & P(y)) (\ x.ball(x))`
- `\ y.  (red(y) & \ x.ball(x)(y))`

## Adjectives

Adjectives are relatively simple unary predicates, but with an added conjunction.

For example, the target semantics for a noun modified by an adjective would be *red ball*, would translate to:

```
\ x.red(x) & ball(x)
```

The result is obtained using this lambda expression for *red*:

```
\ P y.  (red(y) & P(y))
```

- *red ball*
- `\ P y.  (red(y) & P(y)) (\ x.ball(x))`
- `\ y.  (red(y) & \ x.ball(x)(y))`
- `\ y.(red(y) & ball(y))`

## Expression types

**Basic types**

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

## Expression types

### Basic types

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

- **e** is the type for entities

## Expression types

**Basic types**

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

- **e** is the type for entities
- **t** is the type for formulas, i.e., expressions which have truth values (True or False).

## Expression types

### Basic types

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

- **e** is the type for entities
- **t** is the type for formulas, i.e., expressions which have truth values (True or False).

### **e** type

The 'e' stands for entity in the *UD*. Constants and variables (terms) map to entities in the *UD*:

## Expression types

### Basic types

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

- **e** is the type for entities
- **t** is the type for formulas, i.e., expressions which have truth values (True or False).

### **e** type

The 'e' stands for entity in the *UD*. Constants and variables (terms) map to entities in the *UD*:

- *BILL* is of type **e**

## Expression types

### Basic types

Syntactically speaking, expressions in the FOL+$\lambda$ language come in 2 basic types: **e** and **t**

- **e** is the type for entities
- **t** is the type for formulas, i.e., expressions which have truth values (True or False).

### **e** type

The 'e' stands for entity in the *UD*. Constants and variables (terms) map to entities in the *UD*:

- *BILL* is of type **e**
- $x$ is of type **e**

## Expression types

**t** type

type for formulas, i.e., expressions which have truth values (True or False):

## Expression types

**t** type

type for formulas, i.e., expressions which have truth values (True or False):

- $boy(x)$

## Expression types

**t type**

type for formulas, i.e., expressions which have truth values (True or False):

- $boy(x)$
- $\forall x.smokes(x)$

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

## Expression types

**t** type

type for formulas, i.e., expressions which have truth values (True or False):

- $boy(x)$
- $\forall x.smokes(x)$
- $\exists y.knows(y, BILL)$

## Expression types

**Definition**

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

## Expression types

### Definition

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

### Complex types

There are arbitrarily many complex types expressed by their signatures. The set of types is defined as follows:

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

## Expression types

### Definition

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

### Complex types

There are arbitrarily many complex types expressed by their signatures. The set of types is defined as follows:

- $e$ is a (basic) type

## Expression types

### Definition

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

### Complex types

There are arbitrarily many complex types expressed by their signatures. The set of types is defined as follows:

- $e$ is a (basic) type
- $t$ is a (basic) type

Scott Farrar CLMA, University of Washington far-rar@u.washington.ed

## Expression types

### Definition

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

### Complex types

There are arbitrarily many complex types expressed by their signatures. The set of types is defined as follows:

- $e$ is a (basic) type
- $t$ is a (basic) type
- If $a$ and $b$ are types, then so is $\langle a, b \rangle$.

## Expression types

**Definition**

Now, lets assume that expressions in our FOL can be either functions and arguments, as the $\lambda$ calculus does. Functions have **signatures** which define (1) what kinds of arguments the function takes and (2) the return type.

**Complex types**

There are arbitrarily many complex types expressed by their signatures. The set of types is defined as follows:

- $e$ is a (basic) type
- $t$ is a (basic) type
- If $a$ and $b$ are types, then so is $\langle a, b\rangle$.
- Nothing except the basic types, and what can be constructed from them by means of the previous clause are types.

## Expression types

### Complex types

## Expression types

### Complex types

1. $< e, t >$: signature for unary predicates, ie sets in $UD$

## Expression types

**Complex types**

1. $< e, t >$: signature for unary predicates, ie sets in $UD$
2. $< e, < e, t >>$: signature for binary predicates, ie relations among sets in $UD$

## Expression types

### Complex types

1. $< e, t >$: signature for unary predicates, ie sets in $UD$
2. $< e, < e, t >>$: signature for binary predicates, ie relations among sets in $UD$
3. $< e, < e, < e, t >>>$: signature for a more complex function

## Expression types

### Complex types

1. $< e, t >$: signature for unary predicates, ie sets in $UD$
2. $< e, < e, t >>$: signature for binary predicates, ie relations among sets in $UD$
3. $< e, < e, < e, t >>>$: signature for a more complex function
4. ...

## Expression types

### Complex types

1. $< e, t >$: signature for unary predicates, ie sets in *UD*
2. $< e, < e, t >>$: signature for binary predicates, ie relations among sets in *UD*
3. $< e, < e, < e, t >>>$: signature for a more complex function
4. ...

### $< e, t >$

$< e, t >$ means that some function takes something of type **e** and returns something of type **t**. For instance, a unary predicate is one such example.

## Complex Types

**$< e, < e, t >>$**

$< e, < e, t >>$ means that some expression takes something of type **e** and returns something of type **¡e,t¿**. For instance, a binary predicate is one example.

**$< e, e >$**

What about this one?

## Complex Types

**$< e, < e, t >>$**

$< e, < e, t >>$ means that some expression takes something of type **e** and returns something of type **¡e,t¿**. For instance, a binary predicate is one example.

**$< e, e >$**

What about this one?
Consider the named function `father(x)`.

## Complex Types

**$< e, < e, t >>$**

$< e, < e, t >>$ means that some expression takes something of type **e** and returns something of type **¡e,t¿**. For instance, a binary predicate is one example.

**$< e, e >$**

What about this one?
Consider the named function `father(x)`.
`fatherJOHN` results in `TED`

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

## Universal quantifier

The quantifier words *every* and *all* translate to the universal quantifier $\forall$, plus a conditional, e.g., *All CEOs smoke.*

$$\forall x(CEO(x) \rightarrow smoke(x))$$

We need to ensure that the structure of this quantifier phrase gets preserved. The attachment for *all* is:

```
\ P Q. all x.  (P (x) -> Q (x))
```

## Universal quantifier example

*All boys smoke.*

## Universal quantifier example

*All boys smoke.*

1. `\ P Q. all x.  (P (x) -> Q (x)) (\ z. boy(z))(\ s.smoke(s))`

## Universal quantifier example

*All boys smoke.*

1. `\ P Q. all x.  (P (x) -> Q (x)) (\ z. boy(z))(\ s.smoke(s))`
2. `\ Q. all x.  ((\ z.  boy(z)) (x) -> Q (x))`

## Universal quantifier example

*All boys smoke.*

1. `\ P Q. all x.  (P (x) -> Q (x)) (\ z. boy(z))(\ s.smoke(s))`
2. `\ Q. all x.  ((\ z.  boy(z)) (x) -> Q (x))`
3. `\ Q. all x.  (boy(x) -> Q (x))`

## Universal quantifier example

*All boys smoke.*

1. `\ P Q. all x.  (P (x) -> Q (x)) (\ z. boy(z))(\ s.smoke(s))`
2. `\ Q. all x.  ((\ z.  boy(z)) (x) -> Q (x))`
3. `\ Q. all x.  (boy(x) -> Q (x))`
4. `\ Q. all x.  (boy(x) -> Q (x))(\ s.smoke(s))`

## Universal quantifier example

*All boys smoke.*

```
1 \ P Q. all x.  (P (x) -> Q (x)) (\ z.
  boy(z))(\ s.smoke(s))
2 \ Q. all x.  ((\ z.  boy(z)) (x) -> Q (x))
3 \ Q. all x.  (boy(x) -> Q (x))
4 \ Q. all x.  (boy(x) -> Q (x))(\ s.smoke(s))
5  all x.  (boy(x) -> \ s.smoke(s) (x))
```

## Universal quantifier example

*All boys smoke.*

1. `\ P Q. all x. (P (x) -> Q (x)) (\ z. boy(z))(\ s.smoke(s))`
2. `\ Q. all x. ((\ z. boy(z)) (x) -> Q (x))`
3. `\ Q. all x. (boy(x) -> Q (x))`
4. `\ Q. all x. (boy(x) -> Q (x))(\ s.smoke(s))`
5. `all x. (boy(x) -> \ s.smoke(s) (x))`
6. `all x. (boy(x) -> smoke(x))`

# NLTK notes for hw6

For hw6 use a feature context free grammar to parse the simple sentences. Notice how the func-arg relation is represented here:

```
% start S

S[sem = <?subj(?vp)>] -> NP[sem=?subj] VP[sem=?vp]
...
IV[sem=<\x.run(x)>] -> 'runs'
...
```

Just use simple semantic attachments, no m/s features required. Try event semantics if you want.

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