Lambda Calculus for Language Modeling
Day Two: Syntax

Greg Kobele and Sylvain Salvati

23. June 2014
NASSLLI 6
Monday  Intro to $\lambda$-calculus
Tuesday  Using $\lambda$-calculus for syntax (I)
Wednesday Using $\lambda$-calculus for syntax (II)
Thursday Models of the $\lambda$-calculus
Friday  Using $\lambda$-calculus for semantics
Broad Overview

- Why the $\lambda$-calculus?
- Why model language with it?
<table>
<thead>
<tr>
<th>Languages of</th>
<th>Alphabet</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings</td>
<td>$\Sigma = {a_1, \ldots, a_k}$</td>
<td>$\Sigma^*$</td>
</tr>
<tr>
<td>Trees</td>
<td>$\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_k$</td>
<td>$T_\Sigma$</td>
</tr>
<tr>
<td>Lambda Terms</td>
<td>$\Sigma = \langle A, C, \tau \rangle$</td>
<td>$\Lambda(\Sigma)$</td>
</tr>
</tbody>
</table>
## Vocabularies

<table>
<thead>
<tr>
<th>Languages of</th>
<th>Alphabet</th>
<th>Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strings</td>
<td>$\Sigma = {a_1, \ldots, a_k}$</td>
<td>$\Sigma^*$</td>
</tr>
<tr>
<td>Trees</td>
<td>$\Sigma = \Sigma_0 \cup \cdots \cup \Sigma_k$</td>
<td>$T_\Sigma$</td>
</tr>
<tr>
<td>Lambda Terms</td>
<td>$\Sigma = \langle A, C, \tau \rangle$</td>
<td>$\Lambda(\Sigma)$</td>
</tr>
</tbody>
</table>
(Simple) Types

\[ a \in A \]

\[ \alpha, \beta \in T(A) := a \]

\[ | (\alpha \rightarrow \beta) \]

\[ ((e \rightarrow t) \rightarrow t) \rightarrow e \rightarrow t \]
(Simple) Types

\[ a \in A \]

\[ \alpha, \beta \in T(A) := a \]

\[ | (\alpha \rightarrow \beta) \]

\[ ((e \rightarrow t) \rightarrow t) \rightarrow e \rightarrow t \]
(Simple) Types

\[
a \in A
\]
\[
\alpha, \beta \in T(A) := a \\
\quad | (\alpha \rightarrow \beta)
\]
\[
((e \rightarrow t) \rightarrow t) \rightarrow e \rightarrow t
\]
(Simple) Types

\[ a \in A \]
\[ \alpha, \beta \in T(A) := a \]
\[ | (\alpha \rightarrow \beta) \]

\[ ((e \rightarrow t) \rightarrow t) \rightarrow e \rightarrow t \]
The order of a type indicates its complexity

\[
\begin{align*}
\text{order}(a) &= 1 \\
\text{order}(\alpha \to \beta) &= \max(\text{order}(\alpha) + 1, \text{order}(\beta))
\end{align*}
\]

\[
((e \to t) \to t) \to e \to t
\]

\[
\begin{array}{c}
t \\
\downarrow \\
t \\
\downarrow \\
et
\end{array}
\]
A higher order signature \( \Sigma = \langle A, C, \tau \rangle \) is:

- \( A \) is a finite set of atomic types
- \( C \) is a finite set of constants
- \( \tau : C \rightarrow \mathcal{T}(A) \)

\[
\text{order}(\Sigma) := \max \{ \text{order}(\tau(c)) : c \in C \}
\]

Second Order HOS (Tree Signature)

all \( c \in C \) have types \( a_1 \rightarrow \cdots \rightarrow a_k \rightarrow a \)
(all constants map objects to objects)
A higher order signature $\Sigma = \langle A, C, \tau \rangle$ is:

- $A$ is a finite set of atomic types
- $C$ is a finite set of constants
- $\tau : C \rightarrow T(A)$

$$\text{order}(\Sigma) := \max(\{\text{order}(\tau(c)) : c \in C\})$$

**Second Order HOS (Tree Signature)**

all $c \in C$ have types $a_1 \rightarrow \cdots \rightarrow a_k \rightarrow a$

(all constants map objects to objects)
Given HOS $\Sigma = \langle A, C, \tau \rangle$

$\alpha, \beta \in T(A)$

$c \in C$

$M, N \in \Lambda(\Sigma) := x^\alpha$

$\mid c^{\tau(c)}$ (const)

$\mid (M^{\alpha\rightarrow\beta}N^{\alpha})^{\beta}$ (app)

$\mid (\lambda x^\alpha.M^{\beta})^{\alpha\rightarrow\beta}$ (abs)

linearity: Each binder binds exactly one variable
Lambda Terms

- Given HOS $\Sigma = \langle A, C, \tau \rangle$

  $\alpha, \beta \in T(A)$
  $c \in C$

  $M, N \in \Lambda(\Sigma) := x^\alpha$
  $\vert c^\tau(c)$
  $\vert (M^\alpha \rightarrow^\beta N^\alpha)^\beta$
  $\vert (\lambda x^\alpha.M^\beta)^{\alpha \rightarrow^\beta}$

  linearity: Each binder binds exactly one variable
Lambda Terms

Given HOS $\Sigma = \langle A, C, \tau \rangle$

$\alpha, \beta \in \mathcal{T}(A)$

$c \in C$

$M, N \in \Lambda(\Sigma) := x^\alpha$

$\mid c^{\tau(c)}$ (const)

$\mid (M^{\alpha\rightarrow\beta} N^\alpha)^\beta$ (app)

$\mid (\lambda x^\alpha . M^\beta)^{\alpha\rightarrow\beta}$ (abs)

linearity: Each binder binds exactly one variable
Given HOS $\Sigma = \langle A, C, \tau \rangle$

$\alpha, \beta \in T(A)$

$c \in C$

$M, N \in \Lambda(\Sigma) := \begin{array}{ll}
\chi^\alpha & \text{(var)} \\
\tau(c) & \text{(const)} \\
(M^\alpha \rightarrow^\beta N^\alpha)^\beta & \text{(app)} \\
(\lambda x^\alpha M^\beta)^{\alpha \rightarrow^\beta} & \text{(abs)}
\end{array}$

linearity: Each binder binds exactly one variable
Lambda Terms

Given HOS $\Sigma = \langle A, C, \tau \rangle$

$\alpha, \beta \in \mathcal{T}(A)$

$c \in C$

$M, N \in \Lambda(\Sigma) := x^\alpha$

$| c^\tau(c)$

$| (M^\alpha \rightarrow^\beta N^\alpha)^\beta$  (app)

$| (\lambda x^\alpha . M^\beta)^{\alpha \rightarrow^\beta}$  (abs)

linearity: Each binder binds exactly one variable
Lambda Terms

Given HOS \( \Sigma = \langle A, C, \tau \rangle \)

\[ \alpha, \beta \in \mathcal{T}(A) \]
\[ c \in C \]

\[ M, N \in \Lambda(\Sigma) := \begin{cases} \chi^{\alpha} & \text{(var)} \\ c^{\tau(c)} & \text{(const)} \\ (M^{\alpha \rightarrow \beta} N^{\alpha})^{\beta} & \text{(app)} \\ (\lambda x^{\alpha}. M^{\beta})^{\alpha \rightarrow \beta} & \text{(abs)} \end{cases} \]

linearity: Each binder binds exactly one variable
Curry Typed Presentation

\[
\begin{align*}
\text{Ax} & \quad x : \alpha \vdash \Sigma x : \alpha \\
\text{Con} & \quad c \in C \\
\quad \vdash \Sigma c : \tau(c)
\end{align*}
\]
\( \Lambda_\alpha(\Sigma) \), the language of type \( \alpha \) given HOS \( \Sigma \), is

\[
\left\{ M \mid \vdash_\Sigma M : \alpha \right\}
\text{ and } M \text{ in } \eta\text{-long normal form}
\]
Representing a tree as a λ-term

- *Immediate dominance* is represented by application
- *Sibling order* is represented by order of application

```
f -> (f t₁ t₂)
```

\[ t₁ \quad t₂ \]

\[ \rightsquigarrow \] \[ ((f \ t₁) \ t₂) \]
Example

\[ \sim \Rightarrow ( ( f \ ( g \ \ (( h \ a) \ b) \ c)) ) \ d ) \]
Example

\[ f \rightarrow f (g (h \ a \ b \ c)) \ d \]
Assigning types to constants

Rank
The rank of a tree symbol is the number of children it must have

▶ the rank of a leaf is 0
▶ the rank of a unary tree symbol is 1
  
▶ $a^{(n)}$ is a symbol with rank $n$

Tree types

▶ a leaf has no children, it is just a tree: $\tau(a^{(0)}) = t$
▶ a unary branching node has one child, it will be a tree once it takes a tree (its daughter) as input: $\tau(a^{(1)}) = t \rightarrow t$
  
▶ an $n$-ary branching node has $n$ children, it will be a tree once it takes $n$ trees (its children) as input: $\tau(a^{(n)}) = t^n \rightarrow t$
The set of all trees

Given a ranked alphabet $\Sigma$ of node labels,

$$\text{Tree}_\Sigma = \langle \{t\}, \Sigma, \tau \rangle$$

with $\tau(\sigma^n) = t^n \rightarrow t$, for each $\sigma \in \Sigma$

The set of all trees over $\Sigma$ is just $\Lambda_t(\text{Tree}_\Sigma)$
Tree contexts

An element of $\Lambda_{t \rightarrow t}(\text{Tree}_\Sigma)$ is a tree context

$$f \xrightarrow{a \square} (\lambda x. f \ a \ x)$$

A tree context is a tree with a missing subtree

t $\rightarrow$ t

is a second order type
Tree context contexts

An element of $\Lambda_{(t \to t) \to t}(\text{Tree}_\Sigma)$ is a tree context context

$$f \leadsto \lambda x. f \ a \ (x \ b)$$

A tree context context
is a tree with a tree context missing

$(t \to t) \to t$ is a third order type
I will praise Bill.
Adjunction

I will praise Bill with gusto

```
NP | I'  |
  | VP  |
  | NP  |
Susan | I   | V | VP* |
      | will| praise | with |
      | NP  | Bill  | NP  |
      | gusta|
```

IP | VP  |
  | PP  |

```
```
Susan \( \rightarrow \) I \( \rightarrow \) will \( \rightarrow \) VP \( \rightarrow \) NP \( \rightarrow \) with \( \rightarrow \) PP \( \rightarrow \) NP

\( I' \rightarrow V \rightarrow \text{praise} \rightarrow \text{Bill} \)

Adjunction
Adjunction

I will praise Bill with gusto.
Adjunction

I will praise Bill with gusto.
Adjunction

IP
  └── NP
      └── Susan
I
  └── I'
      └── VP
          └── PP
              └── NP
                  └── gusto
V
  └── will
    └── VP
      └── NP
          └── with
            └── Bill
Praise

Adjunction, with lambdas

\[
(\lambda x. IP (NP s) (I' (I w) (x (V p) (NP b)))) : (t \rightarrow t \rightarrow t) \rightarrow t
\]

\[
(\lambda y, z. VP (VP y z) (PP w (NP g))) : t \rightarrow t \rightarrow t
\]
Adjunction, with lambdas

\[
(\lambda x. IP (NP s) (I' (I w) (x (V p) (NP b)))) : (t \rightarrow t \rightarrow t) \rightarrow t
\]

\[
(\lambda y, z. VP (VP y z) (PP w (NP g))) : t \rightarrow t \rightarrow t
\]
Adjunction, with lambdas

\[(\lambda x. \text{IP} \ (\text{NP} \ s) \ (\text{I}' \ (\text{I} \ \text{w}) \ (x \ (\text{V} \ p) \ (\text{NP} \ b)))}) : (t \to t \to t) \to t \]

\[(\lambda y, z. \text{VP} \ (\text{VP} \ y \ z) \ (\text{PP} \ \text{w} \ (\text{NP} \ g))) : t \to t \to t \]
Representing a string as a $\lambda$-term

A string is a unary branching tree:

$$
\begin{array}{c}
a \\
| \\
b \\
| \\
c \\
| \\
# \\
\end{array}
\xrightarrow{\sim} a \ (b \ (c \ #))
$$

As a tree signature, we would have:

$$
\langle \{ t \}, \{ a, b, c, \# \}, \tau \rangle,
$$

with

$$
\tau : \left[ \begin{array}{cc}
a, b, c & \mapsto (t \rightarrow t) \\
\# & \mapsto t \\
\end{array} \right]
$$
Representing a string as a $\lambda$-term

A string is a unary branching tree context.  
- # doesn’t vary across vocabularies  
- we want the strings to be a monoid under concatenation  

$$a 
\begin{array}{c} b \\
\mid \mid \\
\mid \\
\square \\
\end{array} 
\leadsto \lambda x. a \ (b \ (c \ x)) = /abc/$$

$\text{String}_{\Sigma}$

$\langle \{o\}, \Sigma, \tau \rangle$, with  
$\tau(\sigma) = (t \rightarrow t)$  
- $\epsilon := \lambda x. x$  
- $+ := \lambda u, v, x. u \ (v \ x)$
<table>
<thead>
<tr>
<th>Arabic</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td>kitaab</td>
<td>book</td>
</tr>
<tr>
<td>maktab</td>
<td>desk</td>
</tr>
<tr>
<td>kutub</td>
<td>books</td>
</tr>
<tr>
<td>kaatib</td>
<td>writer</td>
</tr>
</tbody>
</table>
Non-local dependencies encodes as *pairs* of strings:

\[
\langle \text{to like Bill, Susan} \rangle \leftrightarrow \langle \text{seems to like Bill, Susan} \rangle \\
\langle \text{seems to like Bill, Susan} \rangle \leftrightarrow \text{Susan seems to like Bill}
\]

**Pairs**

\[
\text{pair} := \lambda x, y, f. f \ x \ y
\]

\[
\langle u, v \rangle = \lambda f. f \ u \ v : ((o \to o) \to (o \to o) \to o \to o) \to o \to o
\]
Define λ-terms that compute the following operations:

\[
\langle s, t \rangle \oplus \langle u, v \rangle = \langle sut, v \rangle \quad (1)
\]
\[
\langle s, t \rangle \otimes \langle u, v \rangle = \langle \mathbf{abc}, su, a \mathbf{t} b \rangle \quad (2)
\]
Subclasses of $\lambda$-terms

We can define the set of all (linear) $\lambda$-terms of a given type but who cares? we want to pick out certain terms as grammatical/meaningful/important

We use types to identify different subclasses

- signatures have had just a single type ($t$ for trees, $o$ for strings)
- we could enrich our atomic types so they would encode more information

$$\Sigma = \{ a^{e \mapsto o}, b^{o \mapsto e} \}$$

$$\epsilon, /ab/, /abab/, \ldots \in \Lambda^o_o(\Sigma)$$
Define the following languages:

- $a^* cb^*$
- derivation trees of the CFG with rules:

$$
S \rightarrow aT
\quad T \rightarrow SbT
\quad T \rightarrow c
$$

e.g.

```
S \rightarrow aT
  |   |
T \rightarrow SbT
  /  /
S \rightarrow aT  T \rightarrow c
    |      |
T \rightarrow c
```
This isn’t enough

- we can describe only a small subclass of regular languages like this
- not even all finite languages (not: \{/aa/\})

An idea:
Let’s *interpret* symbols as operations!

\[
[S \rightarrow aT] \mapsto \lambda x. a \cdot x
\]
We’ve mapped a λ-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}(\{a, b, c\})$)
We’ve mapped a λ-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}({a, b, c})$)
We’ve mapped a $\lambda$-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}({a, b, c})$).
Context-Free Grammars

We’ve mapped a λ-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in String(\{a, b, c\}))
We’ve mapped a $\lambda$-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}(\{a, b, c\})$)
We’ve mapped a λ-term representing a derivation tree (in $\Lambda_\Sigma(\Sigma)$) to one representing its derived string (in $\text{String}(\{a, b, c\})$).
We’ve mapped a $\lambda$-term representing a *derivation tree* (in $\Lambda_s(\Sigma)$) to one representing its *derived string* (in $\text{String}({a, b, c})$).
We've mapped a \( \lambda \)-term representing a derivation tree (in \( \Lambda_s(\Sigma) \)) to one representing its derived string (in \( \text{String}\{a, b, c\}) \)
We’ve mapped a $\lambda$-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}\{\{a, b, c\}\}$).
We’ve mapped a $\lambda$-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}(\{a, b, c\})$).
We've mapped a $\lambda$-term representing a derivation tree (in $\Lambda_s(\Sigma)$) to one representing its derived string (in $\text{String}(\{a, b, c\})$).
We’ve mapped a λ-term representing a derivation tree (in \( \Lambda_s(\Sigma) \)) to one representing its derived string (in \( \text{String}({a, b, c}) \))
Context-Free Grammars

\[
S \rightarrow aT
\]

\[
T \rightarrow SbT
\]

\[
S \rightarrow aT
T \rightarrow c
\]

\[
T \rightarrow c
\]

We’ve mapped a λ-term representing a derivation tree (in \(\Lambda_s(\Sigma)\)) to one representing its derived string (in \(\text{String}\(\{a, b, c\}\)))
We’ve mapped a λ-term representing a derivation tree (in Λ_s(Σ)) to one representing its derived string (in String(\{a, b, c\}))
We’ve mapped a λ-term representing a *derivation tree* (in $\Lambda_\Sigma(\Sigma)$) to one representing its *derived string* (in $\text{String}(\{a, b, c\})$)

\[
[S \rightarrow aT] : t \rightarrow s \quad \mapsto \quad \lambda x.a \cdot x : (oo) \rightarrow (oo)
\]

\[
[T \rightarrow SbT] : s \rightarrow t \rightarrow s \quad \mapsto \quad \lambda x, y.x \cdot b \cdot y : (oo) \rightarrow (oo) \rightarrow (oo)
\]

\[
[T \rightarrow c] : t \quad \mapsto \quad c : (oo)
\]

We are interpreting constants as terms
where the types are substitution instances of each other:

- $t \mapsto o \rightarrow o$
- $s \mapsto o \rightarrow o$
Given $\Sigma_i = \langle A_i, C_i, \tau_i \rangle$, to define a map from $\Sigma_1$ to $\Sigma_2$ we need two things:

- $h : A_1 \rightarrow T(A_2)$
- $g : C_1 \rightarrow \Lambda(\Sigma_2)$

which must satisfy

- $\vdash_{\Sigma_2} g(c) : \hat{h}(\tau(c))$

in particular, $g(c)$ is a closed term.
Homomorphic Extensions

Over types

let \( h : A_1 \rightarrow \mathcal{T}(A_2) \)
define \( \hat{h} : \mathcal{T}(A_1) \rightarrow \mathcal{T}(A_2) \) as follows:

\[
\hat{h}(a) = h(a) \quad \text{(atom)}
\]
\[
\hat{h}(\alpha \rightarrow \beta) = \hat{h}(\alpha) \rightarrow \hat{h}(\beta) \quad \text{(arr)}
\]

Over \( \lambda \)-terms

let \( g : C_1 \rightarrow \Lambda(\Sigma_2) \)
define \( \hat{g} : \Lambda(\Sigma_1) \rightarrow \Lambda(\Sigma_2) \) as follows:

\[
\hat{g}(x) = x \quad \text{(var)}
\]
\[
\hat{g}(c) = g(c) \quad \text{(con)}
\]
\[
\hat{g}(\lambda x. M) = \lambda x. \hat{g}(M) \quad \text{(abs)}
\]
\[
\hat{g}(M N) = \hat{g}(M) \hat{g}(N) \quad \text{(app)}
\]
Lexica and Complexity

Lexicon
We call a pair \( \langle h, g \rangle \) meeting the conditions above a **lexicon** \( L : \Sigma_1 \rightarrow \Sigma_2 \)

Consistency
if \( L : \Sigma_1 \rightarrow \Sigma_2 \), and \( \Gamma \vdash_{\Sigma_1} M : \alpha \), then \( L(\Gamma) \vdash_{\Sigma_2} L(M) : L(\alpha) \)

Complexity
The order of \( L \) is \( \max(\{\text{ord}(h(a)) : a \in A_1\}) \)
Lexicon Composition

**Theorem**

If $\mathcal{L} : \Sigma_1 \rightarrow \Sigma_2$, and $\mathcal{L}' : \Sigma_2 \rightarrow \Sigma_3$, then

$\mathcal{L}' \circ \mathcal{L} = \langle g' \circ g, h' \circ h \rangle : \Sigma_1 \rightarrow \Sigma_3$
Practice

Give lexica which

- compute the yield of a tree
- compute the flip of a tree
- compute the reverse yield of a tree
An ACG $G = \langle \Sigma_1, \Sigma_2, \mathcal{L}, s \rangle$ consists of

- two HOSs $\Sigma_1, \Sigma_2$
- a homomorphism $\mathcal{L} : \Sigma_1 \to \Sigma_2$
- a designated type $s \in \mathcal{T}(A_1)$

The abstract language of $G$: “derivation trees”

$$A(G) := \{ M : \vdash_{\Sigma_1} M : s \}$$

The concrete language of $G$: “derived trees”

$$C(G) := \{ \mathcal{L}(M) : M \in A(G) \}$$
\[ \Lambda_s(\Sigma_1) = A(G) \]

\[ \Lambda_{\hat{h}(s)} \Sigma_2 \]

\[ \mathcal{L} \]
Putting ACGs together
Example TAG: $a^n b^n c^n d^n$

- $S$: Production rules:
  - $S \rightarrow \epsilon$
  - $S \rightarrow S_\emptyset$
  - $S \rightarrow a S d$
  - $S \rightarrow b S_\emptyset c$

Production rules:
- $A_1 = \{s, a\}$
- $A_{stop} : a \rightarrow s$
- $A_{adj} : a \rightarrow a$
- $E : a$

- $s \mapsto (oo)$
- $a \mapsto (oo) \rightarrow (oo)$

Functions:
- $g(A_{stop}) = \lambda f. f \epsilon$
- $g(A_{adj}) = \lambda g, u. a \cdot g (b \cdot u \cdot c) \cdot d$
- $g(E) = \lambda u. u$
Example MCFG: \textit{ww}

\[
\begin{align*}
S(xy) & \ :- \ T(x, y). \quad (\rho_1) \\
T(ax, ay) & \ :- \ T(x, y). \quad (\rho_2) \\
T(bx, by) & \ :- \ T(x, y). \quad (\rho_3) \\
T(\varepsilon, \varepsilon) & \ : \\
\end{align*}
\]

\[
\begin{align*}
A_1 & = \{s, t\} \\
\rho_1 & : t \rightarrow s \\
\rho_2, \rho_3 & : t \rightarrow t \\
\rho_4 & : t
\end{align*}
\]

\textbf{Notation}

\[
(oo) \times (oo) := ((oo)^2 \rightarrow (oo)) \rightarrow (oo)
\]

\[
\begin{align*}
s & \mapsto (oo) \\
t & \mapsto (oo) \times (oo)
\end{align*}
\]

\[
\begin{align*}
g(\rho_1) & = \lambda p. p (\lambda x, y. x \cdot y) \\
g(\rho_2) & = \lambda p, f. p (\lambda x, y. f (a \cdot x) (a \cdot y)) \\
g(\rho_3) & = \lambda p, f. p (\lambda x, y. f (b \cdot x) (b \cdot y)) \\
g(\rho_4) & = \lambda f. f \varepsilon \varepsilon
\end{align*}
\]