Semantics I, Rutgers University
Yimei Xiang

Week 5
October 3, 2018

Lambda Calculus

1. From functions to λ-abstractions

- Recall: A function f from A to B (written as $f\colon A \to B$) is a relation such that (i) f maps every element in A to some element in B, and each element in A is paired with just one element in B.

(1) a. $[\![\text{the mother of}]\!]^w = f\colon D_e \to D_e$ such that for all $x \in D_e$, $f(x)$ is the mother of x in w.
b. $[\![\text{meow}]\!]^w = f\colon D_e \to D_t$ such that for all $x \in D_e$, $f(x) = 1$ iff x meows in w.
c. $[\![\text{invited}]\!]^w = f\colon D_e \to D_{\langle e,t\rangle}$ such that
for all $y \in D_e$, $f(y) = g\colon D_e \to D_t$ such that
for all $x \in D_e$, $g(x) = 1$ iff x invited y in w.

However, if a function is too complex type, the notation could be quite complex. Let's try to define the meaning of *give* using function notations ...

- It is more handy and common to write functions in **λ-notations**. Let's have some intuitions first.

(2) **Schema of lambda abstraction terms:**
$\lambda v[\beta.\alpha]$ or $\lambda v{:}\beta.\alpha$ read as "the function which maps every v such that β to α"
a. λ is called a λ-operator or an abstraction operator
b. v is the argument variable
c. β is the domain condition (the domain over which the function is defined)
d. α is the value description (a specification of the value/output of the function)

For example:

(3) $\lambda x[x \in N.\ x+1]$
read as "the function that maps every x s.t. x is in N to $x+1$."
a. $(\lambda x[x \in N.\ x+1])(2) = 2+1$
$= 3$
b. $(\lambda x[x \in N.\ x+1])(a)$ is undefined

(4) $\lambda x[x \in D_e.\ \mathsf{meow}_w(x)]$
read as "the function that maps every x s.t. x is an entity to x meows in w."
a. $\lambda x[x \in D_e.\ \mathsf{meow}_w(x)](\mathsf{Kitty}) = \mathsf{meow}_w(\mathsf{Kitty})$
b. $\lambda x[x \in D_e.\ \mathsf{meow}_w(x)](\mathsf{grey})$ is undefined

- In semantics, abstraction terms are commonly written as follows:

(5) $\lambda v_\tau.\alpha$ [τ is the semantic type of the variable v]

For example:

(6) a. $[\![\text{meow}]\!]^w = \lambda x_e.\mathsf{meow}_w(x)$
b. $[\![\text{hit}]\!]^w = \lambda y_e.\lambda x_e.\mathsf{hit}_w(x, y)$

2. Lambda calculas

- Lambda (λ-)calculus is a formal system in mathematical logic for expressing computation based on function abstraction and application using variable binding and substitution. (Wiki)

2.1. Syntax

- **λ-terms**

(7) A recursive definition of well-formed λ-terms
a. Every variable is a λ-term. *variables*
b. If E is a λ-term and v is a variable, then $(\lambda v.E)$ is a λ-term. *abstraction terms*
c. If E_1 and E_2 are λ-terms, and so is (E_1E_2). *application terms*

- **Abbreviation rules**

1. As in propositional logic, we omit parentheses if doing so doesn't create ambiguity:
   – $(\lambda v.E)$ can be simplified as $\lambda v.E$
2. Application associates to the left:
   – $E_1E_2E_3$ means $((E_1E_2)E_3)$
3. A sequence of abstractions can be abbreviated:
   – $(\lambda x.(\lambda y.E))$ can be simplified as $\lambda x.\lambda y.E$ or $\lambda xy.E$
4. The body of an abstraction extends as far right as possible:
   – $\lambda x.MN$ means $\lambda x.(MN)$ not $(\lambda x.M)N$

Exercise: Given the syntax in (7), identify whether each of the following notations is a well-formed λ-term. Then, abbreviate the well-formed ones.

(8) a. x
b. (α)
c. $(\lambda x.y)$
d. $((\lambda x.\alpha)\beta)$
e. $((\lambda x.y)((\lambda x.\alpha)y))$

2.2. "Semantics" of simply typed λ-calculus

- The variety of λ-calculus defined above is untyped. It allows us to define functions such as $\lambda f.ff$.

But, in formal semantics, we usually work with a more restrictive system, called the **simply-typed λ-calculus** (Church 1940). In the simply-typed λ-calculus, every function is *partial*: every function expects its arguments to be of a certain type, and is undefined otherwise. For example, $\lambda x_e.x$ and $\lambda x_{\langle e,t\rangle}.x$ both express identity but are different functions.

(9) **Semantic types of lambda terms**
If v is of type σ and α is of type τ, then $\lambda v.\alpha$ is of type $\langle\sigma, \tau\rangle$.

Exercise: Specify the semantic type of each of the following λ-abstracts.

(10) a. $\lambda f_{\langle e,t\rangle}.\lambda x_e.f(x) \wedge \mathsf{gray}(x)$
b. $\lambda f_{\langle e,t\rangle}.\lambda g_{\langle e,t\rangle}.\exists x[f(x) \wedge g(x)]$

- **λ-abstraction**: abstracting a variable and creating a function

  (11) If v is of type σ and α is of type τ, then $\lambda v.\alpha$ is a function f of type $\langle \sigma, \tau \rangle$ s.t. for any object x of type σ, $f(x) = \alpha_{[x/v]}$

(NB: $\alpha_{[x/v]}$, also written as $\alpha[v{:=}x]$, is like α but every **free** occurrence of v in α is replaced by x. Occurrences of v that are free in α are bound by λv in $\lambda v.\alpha$)

- **λ-reduction**
  - **β-reduction**: applying functions to their arguments.

    (12) $(\lambda v.\alpha)(x) = \alpha_{[x/v]}$

  - **α-equivalence**: changing bound variables.

    (13) $\lambda x.\alpha = \lambda y.\alpha_{[y/x]}$

    In a λ-abstraction, the particular choice of a bound variable doesn't matter. For instance, $\lambda x.x$ and $\lambda y.y$ are α-equivalent, and they both represent the same function — the *identity function*. In contrast, the terms x and y are not λ-equivalent, because they are not bound in a λ-abstraction.
  - **η-equivalence**
    η-equivalence expresses the idea of **extensionality**, namely, two functions are equivalent iff they return the same values for every argument.

    (14) $f = \lambda x[f(x)]$ iff x does not appear free in f.

Exercise: Simplify the following λ-notations.

(15) a. $(\lambda x.\lambda x.R(x,x))(y)(z)$

b. $(\lambda x.\lambda y.R(x,y))(v)(u)$

c. $(\lambda x.\lambda y.R(x,x))(y)$

3. Defining semantic values of lexical items using λ-notations

- Predicates
  - Verbs:

    (16) a. $[\![\text{meow}]\!]^w = \lambda x_e.\textsf{meow}_w(x)$
    b. $[\![\text{invite}]\!]^w = \lambda y_e.\lambda x_e.\textsf{invite}_w(x,y)$

  - Non-verbal predicates:

    (17) a. $[\![\text{cat}]\!]^w =$
    b. $[\![\text{larger than}]\!]^w =$
    c. $[\![\text{from China}]\!]^w =$
    d. $[\![\text{from}]\!]^w =$

  Discussion: We don't use the following notations. Do you know why?

  (18) a. $[\![\text{invite}]\!]^w = \lambda \alpha_{\langle e,e\rangle}.\textsf{invite}_w(\alpha)$
  b. $[\![\text{invite}]\!]^w = \lambda x_e.\lambda y_e.\textsf{invite}_w(x,y)$

- Other functions (for now, let's assume that the sentential connectives apply to truth values)
  - Sentential connectives:

    (19) a. $[\![\text{not}]\!]^w = \lambda p_t.\neg p$
    b. $[\![\text{and}]\!]^w =$
    c. $[\![\text{or}]\!]^w =$
    d. $[\![\text{if}]\!]^w =$

  - Functions over functions

    (20) a. $[\![\text{fast}]\!]^w = \lambda P_{\langle e,t\rangle}.\textsf{fast}_w(P)$
    b. $[\![\text{fast}]\!]^w = \lambda P_{\langle e,t\rangle}.\textsf{fast}_w(\lambda x.P(x))$

  Exercise: Simplify the following formulas and then translate them into good English.

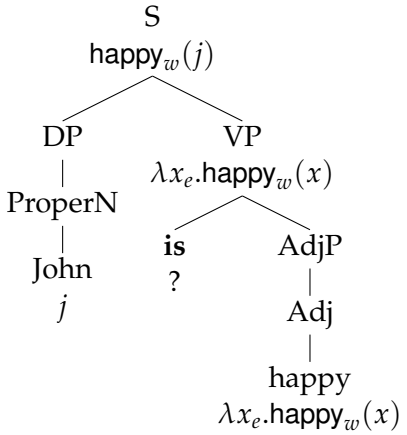
  (21) a. $(\lambda x_e.\lambda y_e[(\lambda z_e.\textsf{introduce-to}_w(x,y,z))(a)])(b)$

  b. $(\lambda f_{\langle e,t\rangle}.\lambda x_e[f(x) \wedge \textsf{gray}_w(x)])(\lambda y_e.\textsf{cat}_w(y))$

  c. $(\lambda P_{\langle e,t\rangle}.P(\textsf{Kitty}))(\lambda y_e.\textsf{cat}_w(y))$

- **Vacuous words**: what is the lexical entry of *is*? We can identify the meaning of an expression based on the meaning of its sister and mother nodes.

(22) John <u>is</u> happy.



Observe that *is* combines with a one-place predicate and returns the same predicate. Hence:

(23) a. $[\![\text{is}]\!]^w = \lambda P_{\langle e,t\rangle}.P$
b. $[\![\text{is}]\!]^w = \lambda P_{\langle e,t\rangle}.\lambda x_e.P(x)$

Exercise: Define the semantics of the article *a* in the following sentence. (Note that *a* is semantically ambiguous. It could also be used as an existential quantifier like *some*, but not in this case.)

(24) Kitty is <u>a</u> cat.

Exercise: Compose the following sentence. How would you define *introduced* and *to*?

(25) Susi introduced Andy to Billy.