CS520, Handout 03, 12 September 2006

The Untyped Lambda Calculus

Reading: [Pierce, Chapter 5]

Syntax

For the sake of further clarity, we follow a notational convention different from Pierce's in the formal syntax: Metavariables (e.g., non-terminal symbols in a BNF grammar) will be in **italics** font, whereas identifiers in the defined language (e.g., terminal symbols in a BNF grammar) will be in **type-writer** font.

Pierce's convention leaves it to the reader to distinguish between a metavariable and a program identifier — see "Variables and Metavariables" at the bottom of [Pierce, page 54].

So, for example, $\boxed{x \text{ is a metavariable}}$ (here ranging over program variables), whereas $\boxed{\texttt{x} \text{ is a program variable}}$. Notice the difference in the fonts.

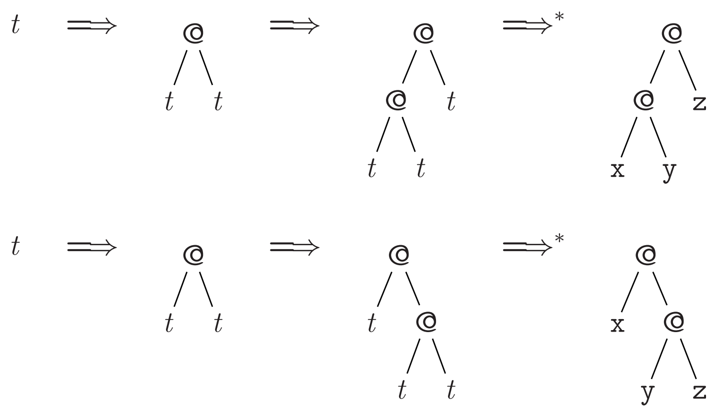
Syntax of the **pure lambda calculus**:

t ::=		*terms*
	x	*variable*
	$\lambda x.\, t$	*abstraction*
	$t\ t$	*application*

This is an **abstract grammar** rather than a **concrete grammar** because, among other things, it does not **uniquely** specify the way in which a term can be generated.

Syntax

For example, the term `x y z` can be parsed (or generated) in two different ways:



If we insert parentheses to disambiguate `x y z`, the first parse corresponds to $\boxed{\texttt{(x y) z}}$ and the second to $\boxed{\texttt{x (y z)}}$.

Syntax

A **concrete grammar** for the terms of the lambda calculus wraps every composite expression with matching parentheses:

$$
\begin{array}{lll}
t ::= & & \textit{terms} \\
 & x & \textit{variable} \\
 & (\lambda x.\, t\,) & \textit{abstraction} \\
 & (t\ t) & \textit{application}
\end{array}
$$

This concrete grammar is **unambiguous** because it generates every term in a unique way.

Syntax

We prefer an abstract grammar (and the abstract syntax it generates) rather than a concrete grammar (and the concrete syntax it generates). This allows us to ignore many syntactic details, e.g., matching pairs of parentheses, that are not essential for understanding semantic aspects of the language.

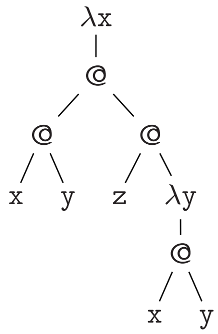
We therefore have to introduce conventions that will allow us to disambiguate terms generated by the abstract syntax:

1. **Applications associate to the left**,
   i.e., $\mathrm{x\,y\,z}$ stands for $(\mathrm{x\,y})\,\mathrm{z}$ not $\mathrm{x}\,(\mathrm{y\,z})$.

2. **Abstractions extend to the right as far as possible**,
   i.e., $\lambda\mathrm{x}.\,\lambda\mathrm{y}.\,\mathrm{x\,y}$ stands for $\lambda\mathrm{x}.\,(\lambda\mathrm{y}.\,\mathrm{x\,y}\,)$ not $\lambda\mathrm{x}.\,(\lambda\mathrm{y}.\,\mathrm{x}\,)\,\mathrm{y}$.

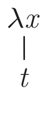
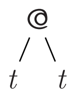
Syntax

Using an **abstract syntax tree** (AST) to represent a term, the term structure is determined by the shape of the tree, and there is no need for parentheses to disambiguate. For example,

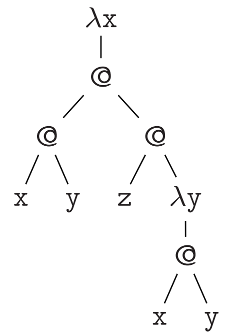


We can use the abstract grammar of the language to generate AST's **unambiguously**, i.e., we think of the abstract grammar as a shorthand for the following concrete grammar for AST's:

Syntax

t ::=		*abstract syntax trees*
	x	*variable*
		*abstraction*
		*application*

Compare the following AST:



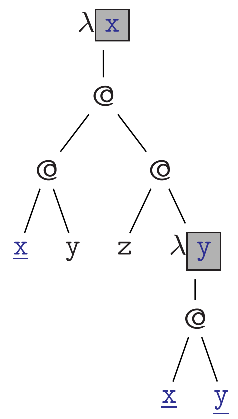
and the corresponding term written as code:

fully parenthesized $(\lambda x.\, ((x\ y)\ (z\ (\lambda y.\, (x\ y)))))$

minimally parenthesized $\lambda x.\, x\ y\ (z\ \lambda y.\, x\ y)$

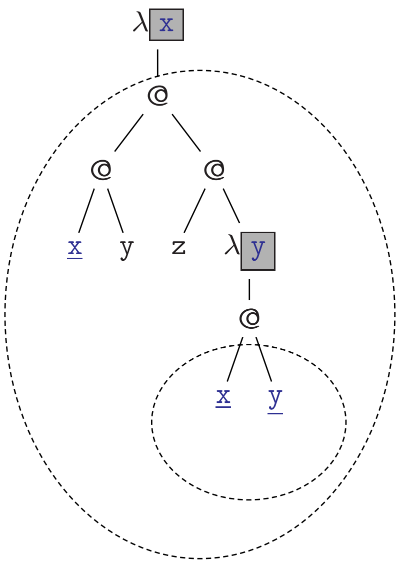
Binding, Bound and Free Occurrences

binding: $\boxed{x}$, bound: $\underline{x}$, free: x.



Scope of Binding Occurrences

Every binding occurrence has a scope.



Semantics of the **pure lambda calculus**, not yet formal.

Only one mechanism for evaluation:

$$\boxed{(\lambda x.\, t\,)\; u \Rightarrow [x \mapsto u]t}$$

The expression on the left $(\lambda x.\, t\,)\; u$ is a term of the lambda calculus, but not the expression on the right $[x \mapsto u]t$.

$[x \mapsto u]t$ refers to an operation on terms. As an expression, it is outside the syntax of the lambda calculus. It means **the term obtained by replacing all free occurrences of x in t by u**.

Clearer but more tedious is to write:

$$\boxed{(\lambda x.\, t\,)\; u \Rightarrow t'}$$

where $t' = [x \mapsto u]t$.

Semantics. Several examples of **beta-reduction**:

$(\lambda x.\, x)\ y \Rightarrow y$

$(\lambda x.\, x)\ \lambda y.\, y \Rightarrow \lambda y.\, y$

$(\lambda x_1.\, \lambda x_2.\, x_1\ x_2)\ (\lambda y.\, y)\ z \Rightarrow (\lambda x_2.\, (\lambda y.\, y)\ x_2)\ z \Rightarrow (\lambda y.\, y)\ z \Rightarrow z$

$(\lambda x_1.\, \lambda x_2.\, x_1\ x_2)\ (\lambda y.\, y)\ z \Rightarrow^* z$ (multistep reduction)

$(\lambda x.\, \lambda y.\, x\ y)\ (y\ z) \Rightarrow \lambda y.\, (y\ z)\ y$ **??** (problem: **variable capture**)

$(\lambda x.\, \lambda y'.\, x\ y')\ (y\ z) \Rightarrow \lambda y'.\, (y\ z)\ y'$ (problem avoided via α-**renaming**)

Semantics. Several evaluation strategies [Pierce, pp 56-57]:

- full beta-reduction
- normal order
- call-by-name
- call-by-need
- call-by-value
- lazy evaluation
- ...

Syntax of the **pure lambda calculus** again,

for call-by-value evaluation:

$$
\begin{array}{rclll}
t & ::= & & & \textit{terms} \\
& & x & & \textit{variable} \\
& & \lambda x.\, t & & \textit{abstraction} \\
& & t\ t & & \textit{application} \\
v & ::= & & & \textit{values} \\
& & \lambda x.\, t & & \textit{abstraction}
\end{array}
$$

Small-Step Call-by-Value Operational Semantics

[Pierce, Sect 5.3]

$$\frac{t_1 \Rightarrow t_2}{t_1\ t \Rightarrow t_2\ t} \qquad \text{(E-App1)}$$

$$\frac{t_1 \Rightarrow t_2}{v\ t_1 \Rightarrow v\ t_2} \qquad \text{(E-App2)}$$

$$(\lambda x.\, t)\ v \Rightarrow [x \mapsto v]t \qquad \text{(E-AppAbs)}$$

Substitution [Pierce, pp 71-72]

$$
\begin{array}{lll}
[x \mapsto s]x & = s & \\
[x \mapsto s]y & = y & \text{if } x \neq y \\
[x \mapsto s](\lambda y.\, t\,) & = \begin{cases} \lambda y.\, t & \text{if } x = y \\ \lambda y.\, [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin FV(s) \end{cases} & \\
[x \mapsto s](t\ u) & = ([x \mapsto s]t)([x \mapsto s]u) &
\end{array}
$$

Glitch: $[x \mapsto y\ z](\lambda y.\, x\ y\,) =$???

Substitution. Under the assumption that bound variables are α-renamed as needed:

$$
\begin{array}{lll}
[x \mapsto s]x & = s & \\
[x \mapsto s]y & = y & \text{if } x \neq y \\
[x \mapsto s](\lambda y.\, t\,) & = \lambda y.\, [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin FV(s) \\
[x \mapsto s](t\ u) & = ([x \mapsto s]t)([x \mapsto s]u) &
\end{array}
$$