COP-5555 PROGRAMMING LANGUAGE PRINCIPLES

NOTES ON LAMBDA-CALCULUS

To obtain the "value" of an RPAL program, we carry out the following steps:

1) Transduce the source PAL program to an AST (using the string-to-tree transduction grammar).
2) Standardize the AST, using the PAL subtree transformational grammar. This step breaks down every construct in RPAL into two basic ones: function abstraction (lambda), and functional application (gamma).
3) Linearize the standardized tree, using the flattener grammar shown below. The result is a lambda-expression.
4) Evaluate the lambda expression. The evaluating mechanism is the Control-Stack-Environment Machine, which we will discuss later. We first need some theory, regarding how lambda-expressions are evaluated.

flattener RPAL:

RPAL → E
E → <'γ' E E > => E E
→ <'λ' V E > => 'λ' V '.' E
→ <id:x> => x
→ <integer:i> => i
→ <string:s> => s
→ 'true' => 'true'
→ 'false' => 'false'
.
.
.
end RPAL.

Note that the result of the transduction is a string (an applicative expression (AE), or well-formed-formula (WFF)). The interpretation of the AE is ambiguous, because there are no parentheses. We will add them to disambiguate, as required by the following rules:

1) Function application is left associative.
2) If an expression of the form λx.M occurs in a larger expression, then M is extended as far as possible (i.e. to the end of the expression or to the next unmatched right parenthesis).

Example: λx.λy.+xy 2 3 is equivalent to λx.(λy.+xy 2 3), and hence must be parenthesized to obtain the intended correct expression: (λx.λy.+xy) 2 3.

Definition: Let M and N be λ-expressions. An occurrence of x in a λ-expression is **free** if it can be proved so via the following three rules:

1) The occurrence of x in λ-expression "x" is free.
2) Any free occurrence of x in either M or N is free in M N.
3) Any free occurrence of x in M is free in λy.M, if x and y are different.

Definition: An occurrence of x in a λ-expression M is said to be **bound** iff it is not free in M.

Examples:

a	-	a occurs free
x	-	x occurs free
a x	-	a and x both occur free
(λx.ax)x	-	a occurs free; x occurs both free and bound

Definition: In an expression of the form λx.M, x is the **bound variable** of the abstraction (the formal parameter name), and M is the **body.**

Definition: The **scope** of an identifier x, in an expression of the form λx.M, consists of all free occurrences of x in M.

Axiom Delta: Let M and N be AE's that do not contain λ-expressions. Then M <=>$_\delta$ N iff Val(M) = Val(N). "Val" is the value obtained from ordinary evaluation. We say that M and N are **delta-convertible**. Example: +35 <=>$_\delta$ 8.

Axiom Alpha: Let x and y be names, and M be an AE with no free occurrences of y. Then, in any context, λx.M <=>$_\alpha$ λy.subst[y,x,M]. "subst[y,x,M]" means "substitute y for x in M", and is defined formally below. This axiom can be used to rename the bound variable. Example: λx.+x3 <=>$_\alpha$ λy.+y3.

Axiom Beta: Let x be a name, and M and N be AE's. Then, in any context, (λx.M) N =>$_\beta$ subst[N,x,M]. This is called a **beta-reduction**, and is used to apply a function to its argument.

Definition: Let M and N be AE's, and x be a name. Then subst[N,x,M] (also denoted as [N/x]M) means

1) If M is an identifier, then

   1.1) if M=x, then return N

   1.2) if M is not x, then return M

2) if M is of the form X Y, then return ([N/x]X)([N/x]Y)

3) If M is of the form λy.Y, then

   3.1) if y=x then return λy.Y

   3.2) if y is not x then

      3.2.1) if x does not occur free in Y, then return λy.Y

      3.2.2) if y does not occur free in N, then return λy.[N/x]Y

      3.2.3) if x occurs free in Y and y occurs free in N, then return λw.[N/x]([w/y]Y), for any w that does nor occur free in either N or Y.

Examples:

a) [3/x](λx.+x2) = λx.+x2 (by 3.1)

b) [3/x](λy.y) = λy.y (by 3.2.1)

c) [3/x](λy.+xy) = λy.[3/x](+xy) = λy.+3y (by 3.2.2 and 2)

d) [y/x](λy.+xy) = λz.[y/x]([z/y](+xy)) = λz.[y/x](+xz) = λz.+yz (by 3.2.3, 2, and 2)

Definition: An AE M is said to be **directly convertible** to an AE N, denoted M <=> N, iff one of these three holds: M <=>$_\alpha$ N, M <=>$_\beta$ N, M <=>$_\delta$ N.

Definition: Two AE's M and N are said to be **equivalent** iff M <=>* N.

Definition: An AE M is in **normal form** iff either

1) M does not contain any λ's, or

2) M contains at least one λ, and

2.1) M is of the form X Y, X and Y are in normal form, and X is not of the form λy.Z.

2.2) M is of the form λx.N, and N is in normal form.

Definition: Given an AE M, a **reduction sequence** on M is a finite sequence of AE's $E_0, E_1, \ldots, E_n$, such that M = E_0 => $E_1 \ldots$ => E_n.

Definition: A reduction sequence is said to **terminate** iff its last AE is in normal form.

Definition: Two AE's M and N are said to be **congruent** iff M $<=>_{\alpha}^{*}$ N.

Definition: A reduction sequence is said to be in **normal order** iff in each reduction, the left-most λ is reduced.

Definition: A reduction sequence is said to be in **programming language order** (PL order) iff for each beta-reduction of the form (λx.M) N, N has been reduced, unless N is a λ-expression.

Examples:

Normal Order:

(λy.y)[(λx.+x3)2] $=>_{\beta}$ (λx.+x3)2 $=>_{\beta}$ (+23) $=>_{\delta}$ 5

PL order:

(λy.y)[(λx.+x3)2] $=>_{\beta}$ (λy.y)(+23) $=>_{\delta}$ (λy.y)5 $=>_{\beta}$ 5

Theorem (Church-Rosser):

1) All sequences of reductions on an AE that terminate, do so on congruent AE's.

2) If there exists a sequence of reductions on an AE that terminates, then reduction in normal order also terminates.

Conclusions:

1) Some applicative expressions can be reduced to normal form, but some cannot (we are still stuck with the classic halting problem).

2) If an AE can be reduced to normal form, then that normal form is unique to within a choice of bound variables.

3) If a normal form exists, a reduction to normal form can be obtained in a finite number of steps using normal order.