Operational Semantics of Pure Functional Languages

Evaluation of an expression in a functional language can be described as a rewriting of the expression into a canonical form, with the function definitions as the rewrite rules.

Rewrite a function application by replacing the function application by the body of the function, substituting the actual arguments for formals, and renaming variables if needed.

Example:

Given the definition

```
double x = x + x
```

Evaluate

```
double 4

double 4 ==>
4+4 ==>
8
```


Evaluation Order

Two important orders of rewriting:

- Normal order - rewrite the leftmost occurence of a function application. *(This is equivalent to call by name.)*
- Applicative order - rewrite the innermost occurrence of a function application first. *(This is equivalent to call by value.)*

Normal order evaluation always gives the same results as lazy evaluation, but may end up evaluating an expression more times.

Examples: Consider

```
double x = x + x
average x y = (x + y) / 2
```

To avoid confusion about infix notation, let's re-express this as:

```
double x = plus x x
average x y = divide (plus x y) 2
```

Evaluate:

```
double (average 2 4)
```


Example of Normal vs. Applicative Order

Using normal order evaluation:

```
double (average 2 4) =>
plus (average 2 4) (average 2 4) =>
plus (divide (plus 2 4) 2) (average 2 4) =>
plus (divide 6 2) (average 2 4) =>
plus 3 (average 2 4) =>
plus 3 (divide (plus 2 4) 2) =>
plus 3 (divide 6 2) =>
plus 3 3 =>
6
```

Notice that `(average 2 4)` was evaluated twice ... lazy evaluation would cache the results of the first evaluation.

Using applicative order evaluation:

```
double (average 2 4) =>
double (divide (plus 2 4) 2) =>
double (divide 6 2) =>
double 3 =>
plus 3 3 =>
6
```


Different Semantics for Normal and Applicative Order Evaluation

Now consider:

```
my_if True x y = x
my_if False x y = y
```

Evaluate:

```
my_if (less 3 4) (plus 5 5) (divide 1 0)
```

Normal order evaluation:

```
my_if (less 3 4) (plus 5 5) (divide 1 0) =>
my_if True (plus 5 5) (divide 1 0) =>
(plus 5 5) =>
10
```

Applicative order evaluation:

```
my_if (less 3 4) (plus 5 5) (divide 1 0) =>
my_if True (plus 5 5) (divide 1 0) =>
my_if True 10 (divide 1 0) =>
DIVIDE BY ZERO ERROR
```

Properties of Evaluation Order; Strictness

Two important properties of evaluation order:

- If there is any evaluation order that will terminate and that will not generate an error, normal order evaluation will terminate and will not generate an error.
- ANY evaluation order that terminates without error will give the same result as any other evaluation order that terminates without error.

Definition: a function f is **strict** in an argument if that argument is always evaluated whenever an application of f is evaluated.

If a function is strict in an argument, we can safely evaluate the argument first if we need the value of applying the function.