Lesson #1: Introduction to Lambda Calculus

Opening Words

1. The λ calculus was devised by Alonzo Church in the 1930s as a model for computability. It is a very simple, very powerful language based on pure abstraction.
2. The λ calculus can be used to formally model aspects of programming languages, and can be viewed as a "machine code" for functional programming languages.
3. Because the λ calculus directly supports abstraction, it more naturally models universal computation than the well-known Turing machine.

Abstraction

Abstraction is a multi-level process of representation based on:

- **generalization¡** through the introduction of a name to replace a value, and
- **specialization** through the replacement of a name with a value.

For example:

- sum of the first 7 natural numbers: 1 + 2 + 3 + 4 + 5 + 6 + 7
- sum of the first 4 natural numbers: 1 + 2 + 3 + 4
- A generalization: sum of the first N natural numbers: 1 + 2 + 3 + ... + N
- A specialization: Replace N with 5 to get 1 + 2 + 3 + 4 + 5

In Lisp:

- `( defun nnsum7 () ( apply #+ ( iota 7 ) ) )`
- `( defun nnsum4 () ( apply #+ ( iota 4 ) ) )`
- `( defun nnsum ( n ) ( apply #+ ( iota n ) ) )`
- Demo:

```
[] ( nnsum4 )
10
[] ( nnsum7 )
28
[] ( nnsum 4 )
10
[] ( nnsum 7 )
28
[] ( nnsum 5 )
15
[] ( nnsum 100 )
5050
```

Lambda Expressions

Definition

The λ **calculus** is a system for manipulating λ *expressions*. A λ **expression** may be a *name*, a *function*, or an *application*.

CFG Description of λ Expressions

The language of lambda expressions is defineded by the following CFG:

1. expression $\longrightarrow$ name | function | application
2. name $\longrightarrow$ a sequence of one or more non-blank characters
3. function $\longrightarrow$ λ name . body
4. body $\longrightarrow$ expression
5. application $\longrightarrow$ (function-expression argument-expression)
6. function-expression $\longrightarrow$ expression
7. argument-expression $\longrightarrow$ expression

Example λ Expressions

1. `x`
2. `(x y)`
3. λ `x . x`
4. λ `first .` λ `second . first`
5. λ `f .` λ `a . ( f a )`

Exercise

Show that the examples just presented are, indeed, lambda expressions by deriving them from the start symbol of the given CFG.

Notes on Functions

1. Functions do not have names!
2. The symbol λ introduces the name used for abstraction. The name is called the function's "bound variable".
3. The body, in which the abstraction takes place, can be any expression.

Notes on Function Application

1. In function application, the function expression is said to be applied to the argument expression.
2. There are two approaches to function application. In both instances, the bound variable of the function expression is replaced throughout the body of the function expression with a variant of the argument expression.
   - Approach 1 - "applicative order" - **call by value** in Algol 60 - the argument expression is evaluated prior to the substitution.
   - Approach 2 - "normal order" - **call by name** in Algol 60 - the argument expression is unevaluated prior to the substitution.

   The latter approach is more general, although it may be less efficient. It, call by name, is the approach that we will use in our examples.

Examples

Consider the function: λ `x . x` (the identity function)

1. We might apply the identity function to the name `whatever`:
   `( λ x ./ x whatever )` $\Longrightarrow$ `whatever`
2. We might apply the identity function to the name `x`:
   `( λ whatever . whatever x )` $\Longrightarrow$ `x`
3. We might apply the identity function to itself:
   `( λ x . x λ x . x )` $\Longrightarrow$ `λ x . x`

Condiser the function: λ `x . ( x x )` (the double function)

1. We might apply the double function to the name `whatever`:
   `λ x . ( x x ) whatever )` $\Longrightarrow$ `( whatever whatever )`
2. We might apply the function to the identity function:
   `( λ x . ( x x ) λ x . x )` $\Longrightarrow$ `( λ x . x λ x . x )` $\Longrightarrow$ `λ x . x`
3. We might apply the function to itself:
   `( λ x . ( x x ) λ x . ( x x ) )` $\Longrightarrow$ `( λ x . ( x x ) λ x . ( x x ) )`
   $\Longrightarrow$ `( λ x . ( x x ) λ x . ( x x ) )` ...