Lambda Calculus

Lambda Calculus Interpreter

Fun to try after you learn the basics of the syntax

Notes

Lambda Calculus

Let's examine some of the theoretical foundations of computation, specifically functional computation.

You may wish to read early parts of this paper:

1. Cardelli and Wegner, "On Understanding Types, Data Abstraction, and Polymorphism"

Notes we will refer to

Encoding Lambda calculus in ML

Boolean values and operators
ML code: booleans

Church numerals
ML code: Church numerals

basic overview

Lambda Calculus as a basis for functional programming languages

More Lambda notes

Yet More Lambda notes

The whole series

And More Lambda notes

Lambda calculus is a formal model of computation. Others include

- Turing Machine
- Post production system (phrase-structure, unrestricted, type-0 grammars)
- Graph/Net models
    - Petri-nets with inhibitor arcs

- predicate/transition nets
- debit-nets under forced anihilation

- Relational model (resolution/unification, Prolog basis)
- Theory of recursive functions

Basics:

- variables are lambda-expressions (lexp)
- lambda abstractions (function definitions) are lexp
- function applications are lexp

For example:

- variables: a, x, foo, bar
- lambda abstraction: lambda x 2*x
- function application: ((lambda x 2*x) 5) specifies value 10

Syntax examples for lambda-expressions

x	a single variable
lambda x x	a function abstraction with one argument (x) and the body "x"
(x y)	function application where function lexp "x" is applied to arg lexp "y"
(lambda x x y)	function "lambda x x" applied to "y"
lambda x (x y)	function abstraction with one variable "x" and body "(x y)" which is a function application
(lambda x x lambda y y)	function application: "lambda x x" is applied to "lambda y y" as an argument
lambda x (x (y x))	function abstraction: body is "(x (y x))" which is an application
lambda x lambda y x	function abstraction defining function of one variable "x" with body "lambda y x" which is another function abstraction.
lambda x lambda y (x lambda x lambda y y)	good and strange

Reductions

Reduction == computation in lambda-calculus

(lambda x M A) can be reduced by substituting A into M for all free occurrances of x.

Examples:

- (L x x (y z)) --> (y z)
- (L x x L x x) --> L x x
- (L x (x y) L z z) --> L z z y --> y
- (L x (x x) L x (x x)) --> (L x (x x) L x (x x)) --> ... nonterminating

Example with abbreviations:

```
((L x L y ((+ x) y) 1) 4)
  --> (L y ((+ 1) y) 4)
  --> ((+ 1) 4)
  --> 5
```

Here we depend on some externally supplied semantics for the symbol "+" which appears where a lambda abstraction must be

```
(L x L y (x y) (y z)) --> ...?? need renaming here to avoid conflicts

naieve approach:

         --> L y ((y z) y)    this "captures" the y in (y z) which
                              was previously unbound

rename:

   (L x L k (x k) (y z)) -->  L k ((y z) k)

which is a different function from L k ((k z) k)
```

Order of evaluation of beta-redexes is important...

Consider this l-exp:

```
( L x (x x) L x (x x) )
```

This expression has no normal form, i.e., there is no way to reduce it so that reduction will terminate.

Now consider this one:

```
( L x y ( L x (x x) L x (x x) ) )
```

Here's how this one works... it is an application of the function

```
L x y
```

to the argument

```
( L x (x x) L x (x x) )   <--- this is a function application
                               that will not terminate... it
                               takes it's argument and replicates
                               it...
```

so if you use an eval order that tries to eval the argument before you call the outermost function you will not terminate.

However, the outermost function ignores it's argument and just returns "y" no matter what the argument is. So normal order eval (outermost, left most first) will reduce fine to "y" as the normal form... applicative order will never reduce.

Normal Order Reduction

(underlining indicated the beta-redex for each step)

```
( L x y ( L x (x x) L x (x x) ) )
  -----
```

so we plug the arg "(L x (x x) L x (x x))" into all occurrances of the bound var "x" in the body of the function "L x y"... and there are no such occurrance so the problematic arg goes away... leaving

```
y
```

Applicative Order Reduction

(underlining indicated the beta-redex for each step)

```
( L x y ( L x (x x) L x (x x) ) )
        ----------------------
```

so we plug the arg "(L x x x)" in for each of the "x" in the body if the function "(L x x x)"... we get

```
( L x y ( L x (x x) L x (x x) ) )
        ----------------------
```

and we have the same problem back... and continue... ad infinitum..

We know these things...

1. some l-exps do not terminate when reduced
2. some l-exps fail to terminate when reduced one way, but reduce successfully when reduced a different way
3. (Church-Rosser) for a given l-exp, all terminating reduction sequences end in the same reduced l-exp
4. If you choose Normal order reduction (outer-most, left-most redex first) you will get a terminating reduction sequence *if one exists*... this models lazy evaluation in functional languages

Booleans

boolean (truth) values are functions of two arguments... "true" returns the first arg, and "false" returns the second arg.

T == L x L y x
F == L x L y y

Then boolean operators can be defined... NOT simple reverses the sense of its arg... so if you do (NOT T) you get F...

NOT == L x ((x F) T)
AND == L x L y ((x y) F)
OR == L x L y ((x T) y)

in class, do XOR

Integers
go over function ZEROP, SUCC, ADD
in class, do MULT

Lists/pairs
from text

Integers

1 == L f L x (f x)
2 == L f L x (f (f x))
N == L f L x (f .... (f x) ...) f applied N times

(N f) is L x (f ... (f x) ...) which is a function abstraction

(N f) b) is (f ... (f b) ...) is an N-fold application, a value

Try this: apply **(M f)** to **((N f) b)**

((M f) ((N f) b))

```
gives ( f ... f ( f ... f ( f b ) ... ) ... ) with M+N f's in the list
        M here     N here
```

ADDITION: M+N == L f L x ((M f) ((N f) x))

ADDITION: + == L A L B L f L x ((A f) ((B f) x))

The second form is fully abstracted for the two numbers being added so it is the operation itself

Addition

M + N == L f L z ((M f) ((N f) z))
+ == L A L B L f L z ((A f) ((B f) z))

Mult

M*N == L z (M (N z))
***** == L A L B L z (A (B z))

Exponentiation

A ^ B == (B A)
^ == L A L B (B A)

Pairs

pair == L a L b L f (f (a b))

head == L g (g (L a L b a))

tail == L g (g (L a L b b))

L-exp to try:

- XOR
- one that lengthens... maybe triples
  Lx (x (x x)) Lx (x (x x))
- one that differs in semantics if evaled L-R vs. R-L