CSCI 275: Programming Abstractions

Lecture 32: Learning a Language

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Goal for the next few days

`(lambda (x y) (+ x y)))`

1. Where does the `lambda` keyword actually come from?

2. Why does Racket's syntax look the way it does?

3. A bunch of other cool things

The content in these lectures is adapted from "Types and Programming Languages" by Pierce, Cornell's CS 4110/6110 Notes and Phil Wadler's "Propositions as Types"

MiniScheme

In the MiniScheme project, we wrote an **interpreter** for a language called MiniScheme

- MiniScheme has a **formal grammar** that we wrote down
- We made **parse trees** to represent an intermediate version of the language
- We then interpreted those parse trees to **evaluate MiniScheme expressions**

Learning a Language & Practical Concerns

What I want you to take away from this class is a practiced, defined notion of

Language design and implementation fundamentals

What's a good way to learn a language?

Know the most *fundamental* underlying structure!

To Spoil the Punchline….

The rest of this week we are going to talk about the first programming language

It’s called the ***lambda calculus***

Invented in 1935 by Alonzo Church

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 349 December 1975

SCHEME

AN INTERPRETER FOR EXTENDED LAMBDA CALCULUS

by

Gerald Jay Sussman and Guy Lewis Steele Jr.

Abstract:

Inspired by ACTORS [Greif and Hewitt] [Smith and Hewitt], we have implemented an interpreter for a LISP-like language, SCHEME, based on the lambda calculus [Church], but extended for side effects, multiprocessing, and process synchronization. The purpose of this implementation is tutorial. We wish to:

Introduction to the Lambda Calculus

The Lambda Calculus

Much like other languages, the lambda calculus has a *syntax* and a *semantics*. Here is its syntax:

e :: =	x	*variable*
	λx. e	*function abstraction*
	e_1 e_2	*function application*

Use parentheses for grouping terms together (λx. λy. x) a b

Function application is left associative: f x y is the same as (f x) y

How do we compute with this?

It is *very simple*: all we can do in the base lambda calculus is apply functions to arguments.

Examples:

`(λx. x) a` gives `a`

`(λx. x (λx. x)) b` gives us `b (λx. x)`

How do we compute with this?

It is *very simple*: all we can do in the base lambda calculus is apply functions to arguments.

Substituting arguments into functions is called *beta-reduction*

Examples:

`(λx. x) a` gives `a`

`(λx. x (λx. x)) b` gives us `b (λx. x)`

These terms are called *reducible expressions*

How do we compute with the lambda calculus?

We can actually write *many more meaningful* programs than you might expect!

Church Booleans

Church Numerals

Reminder: Currying

Currying is the approach of returning a function from another function:

```
(define equal-x-checker
  (lambda (x)
    (lambda (y)
        (equal? y x))))
```

`Then (equal-x-checker 3)` will be a procedure that checks whether any input is equal to 3

`((equal-x-checker 3) 4) is #f`

Currying is *default* in the lambda calculus

Curried functions are actually the only multi-argument functions in the lambda calculus:

$$\lambda x.\ \lambda y.\ y$$

We could add something like below, but we choose not to:

$$\lambda xy.\ y$$

Church Booleans

We can encode values for true and false. We call these “Church Booleans”

Intuition: true and false are two argument functions; they act like `(if #t t f)` and `(if #f t f)` in Scheme

```
true t f = t
false t f = f
```

Church Booleans

Rewriting these in lambda calculus

```
 true = λt. λf. t
false = λt. λf. f
```

Variable names don't matter!

Encoding And

```
and = λb. λc. b c false
```

Let's walk through the fact this works on the board !

```
true = λt. λf. t
false = λt. λf. f
```

```
If
 true = λt. λf. t
false = λt. λf. f
```

Remember we defined previously as

`and = λb. λc. b c false`

Is there another way to encode `and`?

A. `λb. λc. b c c`

B. `λb. λc. b c b`

C. `λb. λc. b c true`

D. Something else

E. Nope, only one `and`!

Church Numerals

We can also encode numbers in the lambda calculus

Intuition: We'll encode numbers as repeated applications of a function f to a value x

Think of each number as a two argument function that applies its first argument to its second argument that number of times

```
zero f x = x
one f x = f x
two f x = f (f x)
three f x = f (f (f x))
```

Church Numerals

Rewriting this in lambda calculus gives

```
zero = λf. λx. x
 one = λf. λx. f x
 two = λf. λx. f (f x)
   n = λf. λx. f (f …(f x)…)
```

Wait. If

```
 false = λt. λf. f
and
 zero = λf. λx. x
```

Is this a problem?

A. Yes
B. No because they have different types (false is a Boolean and zero is a number)
C. No because they have different parameters
D. No because we can use the same function in different contexts to do different things

Given `one`, how can we get `two`?

We can define a successor function:

```
 one = λf. λx. f x
succ = λn. λf. λx. f (n f x)
```

To get:

```
 two =  λf. λx. f (f x)
```

Let’s try it out:

https://capra.cs.cornell.edu/lambdalab/

How can we add two numbers together?

Given two numbers `n` and `m`, discuss in your small groups how you might intuitively compute `n + m` with just the successor function.

How can we add two numbers together?

One way: given `m`, apply the successor function m times to `n`!

```
plus = λm. λn. n succ m
```

Let’s try it out!

How can we write a recognizer?

Let’s write a recognizer (something that returns a Boolean): `iszero`

This should return (our definition) of `true` if the argument is `zero`, and `false` otherwise

Bonus stuff: Lists

Let’s implement lists in the lambda calculus

We need:

- cons — creating a pair
- fst — car in Scheme
- snd — cdr in Scheme
- null — the empty list
- isnull — null? in Scheme

The “easy” stuff: Pairs

For Church Booleans, we decided to use two-argument functions that returned their first (for true) or second (for false) arguments

We have a similar situation where there are two parts to the pair and we want fst to return the first element of the pair and snd to return the second element

For Church pairs, let’s define the pair as a function that takes a two-argument function and applies that to the two parts of the pair →

Pairs

```
cons = λx. λy. λf. f x y
```

- `Ex. cons (a b) c → λf. f (a b) c`

```
fst = λp. p true        # fst (cons x y) → x
snd = λp. p false       # snd (cons x y) → y
```

From pairs to lists (Tricky!)

A list is either a pair that we get from cons x y or is null

Tricky definition:

```
null = false
isnull = λp. p _____ true
```

- ```
  isnull null = (λp. p _____ true) null
                → null _____ true
                = false _____ true
                → true        (because false x y → y)
  ```
  ```

isnull

`isnull = λp. p _____ true`

What if p is not null? What if it's cons x y?

```
cons x y → λf. f x y
isnull (λf. f x y )
  = (λp. p _____ true) (λf. f x y )
  → (λf. f x y ) _____ true
  → _____ x y true
  → false
```

```
isnull (λf. f x y )
  = (λp. p _____ true) (λf. f x y )
  → (λf. f x y ) _____ true
  → _____ x y true
  → false
```

What can we replace the _____ with such that the final reduction is correct? Work on this in groups and when you have a solution, select any answer

Lists

```
cons = λx. λy. λf. f x y
fst = λp. p true
snd = λp. p false
null = false
isnull = λp. p (λx. λy. λz. false) true
```