EECS 490 – Lecture 12

Lambda Calculus II

10/12/17

Announcements

- HW3 due Fri 10/20
- Project 3 due Fri 10/27

Review: λ-Calculus

- Context-free grammar:

  Expression → *Variable*
  | λ *Variable* . *Expression* **(function abstraction)**
  | *Expression Expression* **(function application)**
  | (*Expression*)

- Variables denoted by single letters

- Examples:

  `λx. x` (identity function)
  `(λx. x) y` (identity function applied to variable *y*)

Review: α-Reduction

- In `(λx. E)`, replacing all occurrences of x with *y* does not change the meaning as long as *y* does not appear in *E*

  `λx. x x` $=_\alpha$ `λy. y y`

- This renaming is called *α-reduction*

- Two expressions are *α-equivalent* if they only differ by α-reductions

Review: β-Reduction

β-equivalent to identity function

- In function application, we apply α-reduction to ensure that the function and its argument have distinct names
  - Accomplishes the same thing as frames and environments

$$(\lambda x.\ x)\ (\lambda x.\ x) \rightarrow_{\alpha} (\lambda x.\ x)\ (\lambda y.\ y)$$

- We then substitute the argument expression for the parameter in the scope of the parameter
  - This is *β-reduction* and is equivalent to a call-by-name parameter-passing strategy

$$(\lambda x.\ x)\ (\lambda y.\ y) \rightarrow_{\beta} \lambda y.\ y$$

- Two expressions are *β-equivalent* if they β-reduce to the same thing

Encoding Data

- Lambda calculus consists only of variables and functions
  - We can apply β-reduction to substitute functions for variables
- None of the familiar values, such as integers or booleans, exist directly in λ-calculus
- However, we can encode values as functions

Booleans

- True and false are represented as functions that take in a true and a false value and return the appropriate value

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

Picks the first value

Picks the second value

Mathematical definition, not assignment

- Logical operators are defined as follows:

```
and = λa. λb. a b a
or = λa. λb. a a b
not = λb. b false true
```

Conjunction

```
true = λt. λf. t
false = λt. λf. f
and = λa. λb. a b a
```

- Applying *and* to *true* and another boolean results in:

```
and true bool = ((λa. λb. a b a) true) bool
              → (λb. true b true) bool
              → (λb. b) bool
              → bool
```

- Applying *and* to *false* and another boolean results in:

```
and false bool = ((λa. λb. a b a) false) bool
               → (λb. false b false) bool
               → (λb. false) bool
               → false
```

Disjunction

```
true = λt. λf. t
false = λt. λf. f
or = λa. λb. a a b
```

- Applying *or* to *true* and another boolean results in:

```
or true bool = ((λa. λb. a a b) true) bool
             → (λb. true true b) bool
             → (λb. true) bool
             → true
```

- Applying *or* to *false* and another boolean results in:

```
or false bool = ((λa. λb. a a b) false) bool
              → (λb. false false b) bool
              → (λb. b) bool
              → bool
```

Negation

```
true = λt. λf. t
false = λt. λf. f
not = λb. b false true
```

- Applying *not* to a boolean results in:

```
not true = (λb. b false true) true
         → true false true
         → false

not false = (λb. b false true) false
          → false false true
          → true
```

Conditional

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

- A conditional takes in a boolean, a "then" value, and an "else" value

```
if = λp. λa. λb. p a b
```

- Applying *if* to *true* and *false* results in:

```
if true x y = (λp. λa. λb. p a b) true x y
            → (λa. λb. true a b) x y
            → (λa. λb. a) x y
            → x

if false x y = (λp. λa. λb. p a b) false x y
             → (λa. λb. false a b) x y
             → (λa. λb. b) x y
             → y
```

Pairs

- A pair is represented as a higher-order function that takes in two items and a function, then applies its function argument to the two items

```
pair = λx. λy. λf. f x y
pair a b = (λx. λy. λf. f x y) a b
         → λf. f a b
```

- We can define selectors:

```
first = λp. p true
second = λp. p false
```

- We can define nil and a null predicate:

```
nil = λx. true
null = λp. p (λx. λy. false)
```

Selectors

```
pair a b → λf. f a b
first = λp. p true
second = λp. p false
```

- Selectors work as follows:

```
first (pair a b) = (λp. p true) (pair a b)
                 → (pair a b) true
                 = (λf. f a b) true
                 → true a b
                 → a

second (pair a b) = (λp. p false) (pair a b)
                  → (pair a b) false
                  = (λf. f a b) false
                  → false a b
                  → b
```

Null Predicate

```
pair a b → λf. f a b
nil = λx. true
null = λp. p (λx. λy. false)
```

- The null predicate works as follows:

```
null nil = (λp. p (λx. λy. false)) λx. true
         → (λx. true) (λx. λy. false)
         → true

null (pair a b) = (λp. p (λx. λy. false)) (pair a b)
                → (pair a b) (λx. λy. false)
                = (λf. f a b) (λx. λy. false)
                → (λx. λy. false) a b
                → (λy. false) b
                → false
```

Trees

- Now that we have pairs, we can represent arbitrary data structures, including trees:

```
tree = λd. λl. λr. pair d (pair l r)
datum = λt. first t
left = λt. first (second t)
right = λt. second (second t)
empty = nil
isempty = null
```

Break Time!

- We'll start again in five minutes.

Church Numerals

- A natural number n is represented as a function that takes in another function f and an item x and applies f to the item a total of n times:

```
zero = λf. λx. x
one = λf. λx. f x
two = λf. λx. f (f x)
three = λf. λx. f (f (f x))
four = λf. λx. f (f (f (f x)))
five = λf. λx. f (f (f (f (f x))))
...
```

- Mathematically speaking, a number n takes f and produces the self-composition f^n

$$\text{three}\, f \rightarrow f^3 = f \circ f \circ f$$

Increment

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

- We can increment a number as follows:

```
incr = λn. λf. λy. f (n f y)

incr zero = (λn. λf. λy. f (n f y)) zero
          → λf. λy. f (zero f y)
          = λf. λy. f ((λx. x) y)
          → λf. λy. f y
          = one

incr one = (λn. λf. λy. f (n f y)) one
         → λf. λy. f (one f y)
         = λf. λy. f ((λx. f x) y)
         → λf. λy. f (f y)
         = two
```

```
two = λf. λx. f (f x)
incr = λn. λf. λy. f (n y)
```


Addition and Multiplication

- We can define addition as follows:

```
plus = λm. λn. m incr n
```

Apply the *incr* function *m* times to *n*

```
plus two three = (λm. λn. m incr n) two three
               → (λn. two incr n) three
               = (λn. (λf. λx. f (f x)) incr n) three
               → (λn. (λx. incr (incr x)) n) three
               → (λx. incr (incr x)) three
               → incr (incr three)
               → incr four
               → five
```

- We can similarly define multiplication:

```
times = λm. λn. m (plus n) zero
```

Apply the *(plus n)* function *m* times to *zero*

We can define exponentiation using the same strategy.

Zero Predicate

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

- Predicate to check for zero:

```
iszero = λn. n (λy. false) true
```

Only results in true if function never applied

```
iszero zero = (λn. n (λy. false) true) zero
            → zero (λy. false) true
            = (λf. λx. x) (λy. false) true
            → (λx. x) true
            → true

iszero one = (λn. n (λy. false) true) one
           → one (λy. false) true
           = (λf. λx. f x) (λy. false) true
           → (λx. (λy. false) x) true
           → (λx. false) true
           → false
```

Recursion

- Functions are anonymous, so need to arrange to pass in function as an argument to itself

```
fact = λf. λn. if (iszero n)
                  one
                  (times n (f f (decr n)))
```

Decrement function

Pass along function to itself in recursive call

- We also need an auxiliary apply function

```
apply = λg. g g
```

Example

- Computing factorial:

```
apply fact m = (λg. g g) fact m
             → fact fact m
             = (λf. λn. if (iszero n) one
                          (times n (f f (decr n))))
               fact m
             → (λn. if (iszero n) one
                        (times n (fact fact
                                       (decr n))))
               m
             → if (iszero m) one
                  (times m (fact fact (decr m)))
             =β if (iszero m) one
                   (times m (apply fact (decr m)))
```

Y Combinator

- Also known as a *fixed-point combinator*

```
Y = λf. (λx. f (x x)) (λx. f (x x))
```

- Applying *Y* to a function *F* results in

```
Y F = (λf. (λx. f (x x)) (λx. f (x x))) F
    → (λx. F (x x)) (λx. F (x x))
    → (λx. F (x x)) (λy. F (y y))
    → F ((λy. F (y y)) (λy. F (y y)))
    = F (Y F)
```

Simpler Factorial

No longer have to pass function to itself

- First, define the concrete function *F*:

```
F = λf. λn. if (iszero n) one (times n (f (decr n)))
```

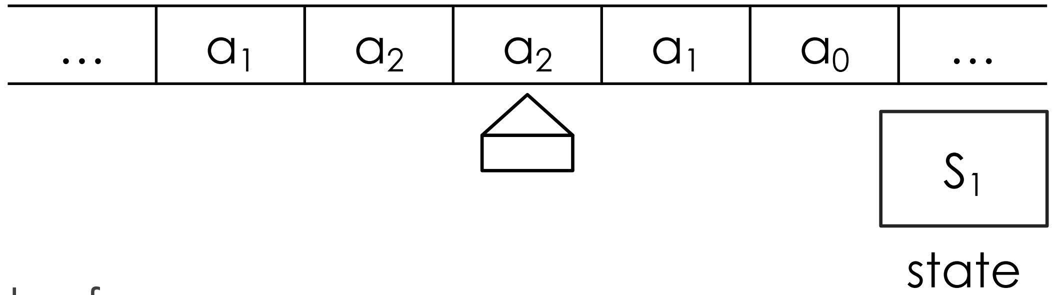
- Now apply *Y* to *F*, and apply result to a number:

```
Y F m → F (Y F) m
      = (λf. λn. if (iszero n) one (times n (f (decr n))))
        (Y F) m
      → (λn. if (iszero n) one (times n (Y F (decr n)))) m
      → if (iszero m) one (times m (Y F (decr m)))
```

- Letting *fact* = *Y F*, we get

```
fact m → if (iszero m) one (times m (fact (decr m)))
```

Turing Machine



- Consists of:
  - An infinite *tape* divided into cells, each containing a symbol from a finite alphabet
  - A *head* positioned at a cell and that can move left or right
  - A *state register* that keeps track of the state of the machine, one of finitely many
  - A *table* of instructions that specifies what to do given a state and the symbol currently under the head
    - Write a specific symbol to the current cell
    - Move the head one step to the left or right
    - Go to a new state

Church-Turing Thesis

- Alan Turing proved that Turing machines solve the same set of problems as λ-calculus
- The Church-Turing thesis states that all problems that can be solved by a human using an algorithm can also be solved on a Turing machine
- All known computational models are weaker than or equivalent to Turing machines
  - Equivalent models are *Turing complete*
- A programming language defines a computational model
  - All general-purpose programming languages are Turing complete