Lecture 9 — August 31, 2015

- Today:
  - Declarative Programming
  - Unification
  - Lambda calculus
- Readings:
  - Finish *SICP* 4.4
  - http://docs.racket-lang.org/datalog/datalog.html

CSE130, Summer Session II

Logic Programming

- Different way to program: No procedures!
- Math: ***Declarative Knowledge****:* What Is
  - The definition of a square root:

$$\sqrt{x} \text{ is the } y \text{ such that } y^2 = x \text{ and } y \geq 0$$

- CS: ***Imperative Knowledge****:* How To
  - The algorithm to compute a square root:

```
(define (sqrt x)
  (let loop ([guess 1.0])
    (if (< (abs (- (square guess) x)) 0.00001)
        guess
        (loop (average guess (/ x guess))))))

(define (square n) (* n n))
(define (average a b) (/ (+ a b) 2))
```

Logic Programming

- The relation between Celsius and Fahrenheit:

  $$F - 32 = (9/5) \cdot C$$

- This formula provides enough information to let us know how to:
  - Convert from Fahrenheit to Celsius; and
  - Convert from Celsius to Fahrenheit
- No explicit input or output:
  - One piece of declarative knowledge can be used as the basis of several kinds of how-to knowledge
  - E.g., in procedural programming, the **`sqrt`** procedure maps inputs to outputs: *"What is the square root of 289?"* but you can't use it to ask *"What is 17 the square root of?"*

Facts and Queries

- Idea: Why not program in declarative terms instead?
  - Specify facts (what is true) an let the system figure out the “how to” part
- For example, queries over a Prolog/Datalog database:

```
% Facts:
son_of(adam, abel).
son_of(adam, cain).
son_of(cain, enoch).
son_of(enoch, irad).
```

```
% Prolog/Datalog Queries:
?- son_of(adam, Who).          % Who is Adam the father of?
Who = abel
Who = cain

?- son_of(Who, cain).          % Who is Cain's father?
Who = adam
```

- Variables start with Uppercase letters.
- Atoms start with lowercase letters (an atom is like a `'symbol` in Lisp)

Prolog/Datalog Rules

- Instead of stating all facts directly, we can ***infer*** other facts from logical rules:

```
% Facts
son_of(adam, abel).
son_of(adam, cain).
son_of(cain, enoch).
son_of(enoch, irad).

% Rules
grandson_of(X, Z) :- son_of(X, Y), son_of(Y, Z).
```

here, **X**, **Y**, and **Z** are the variables

- Now we can ask:

```
?- grandson_of(adam, Who).  % Who is the grandson of Adam?
Who = enoch

?- grandson_of(Who, irad).  % Who is Irad the grandson of?
Who = cain
```

- Read the "**:-**" like a backwards implication and the commas as logical conjunction: $(x \text{ sonof } y \wedge y \text{ sonof } x) \Rightarrow x \text{ grandsonof } y$

Prolog/Datalog Rules

- We can specify a general relation implied from the facts and the rules:

```
% Facts
son_of(adam, abel). son_of(adam, cain).
son_of(cain, enoch). son_of(enoch, irad).

% Rules
grandson_of(X, Z) :- son_of(X, Y), son_of(Y, Z).
relation(X, Y, son) :- son_of(X, Y).
relation(X, Y, father) :- son_of(Y, X).
relation(X, Y, grandson) :- grandson_of(X, Y).
relation(X, Y, grandfather) :- grandson_of(Y, X).
```

```
?- relation(enoch, cain, R).     % What is Enoch and Cain's
R = father                       % relation?

?- relation(_, X, grandfather). % Who are all of the
X = adam                         % grandfathers, based on
X = cain                         % just this information?
```

- The "_" means "don't care"

Lists in Prolog: **`member`**

- Prolog (but not Datalog), supports lists: The built-in **`member`** predicate can be used to make queries:

```
?- member(a, [a, b, c, d]).
true

?- member(z, [a, b, c, d]).
false

?- member(X, [a, b, c, d]).
X = a
X = b
X = c
X = d

?- member(e, [a, b, X, d]).
X = e
```

Lists in Prolog: Cons

- In Prolog, a cons cell is specified using `[Head|Tail]`
- An empty list is specified with `[]`
- The `[Head|Tail]` notation doesn't necessarily *create* a cons cell, it can be used to *retrieve* the head and tail

```
?- A=[1, 2], A=[H|R].
A = [1, 2],
H = 1,
R = [2]

?- H=1, R=[2], A=[H|R].
A = [1, 2],
H = 1,
R = [2]

?- X=[4, 8, 16], [XCar|XCdr]=X, [XCadr|_]=XCdr.
X = [4, 8, 16],
XCadr = 8,
XCar = 4,
XCdr = [8, 16]
```

***A** is bound first: forces **H** and **R** to match*

***H** and **R** are bound first: forces **A** to match*

Lists in Prolog: **member**

- Rules state what is true
- When given variables, Prolog's solver will try to find all matches for the variables that result in truth

```
member(X, [X|_]).
member(X, [_|Tail]) :- member(X, Tail).
```

The first rule says:

- If `X` is the head of the list, then `X` is a member of the list

The second rule says:

- If `X` is a member of the tail of the list, then `X` is a member of the list

```
?- member(X, [1, 2]).
X = 1 ;
X = 2.
```

Lists in Prolog: **append**

- The built-in **append** predicate relates *three* lists:

```
?- append([1], [2], L).
L = [1, 2].
```

```
?- append(A, B, [1, 2, 3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [] ;
false.
```

```
append([], Ys, Ys).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).
```

Expressing the merge process

- In Scheme, we could describe how to ***merge*** the contents of two lists like so:

```
(define (merge x y)
  (cond [(null? x) y]
        [(null? y) x]
        [else (let ([a (car x)]
                    [b (car y)])
                (if (< a b)
                    (cons a (merge (cdr x) y))
                    (cons b (merge x (cdr y)))))]))
```

```
> (merge '(0 2 4 6 8) '(1 3 5 7 9))
'(0 1 2 3 4 5 6 7 8 9)
```

But what is **`merge`**, really?

Let's look into the logic of the program:

```
(cond [(null? x) y]
      [(null? y) x] ...
```

- for all `y`: `'()` and `y` merge to `y`
- for all `x`: `x` and `'()` merge to `x`

```
(let ([a (car x)] [b (car y)])
  (if (< a b)
      (cons a (merge (cdr x) y)) ...
```

- if `(cdr x)` and `y` merge to `Z`
  and `a < (car y)`, then
  `(cons a (cdr x))` and `y` merge to `(cons a Z)`.

Lists in Prolog: **merge**

- `[3, 6, 9]` and `[4, 8]` merge to `What` (`L`)?

```
?- merge([3, 6, 9], [4, 8], L).
L = [3, 4, 6, 8, 9].
```

- `[1, 3, 5]` and `What` merge to `[1, 2, 3, 4, 5]`?

```
?- merge([1, 3, 5], L, [1, 2, 3, 4, 5]).
L = [2, 4].
```

- `What` and `WhatElse` merge to `[1, 2, 3, 4, 5, 6]`?

```
?- merge(What, WhatElse, [1, 2, 3, 4, 5, 6]).
```
This has $2^6 = 64$ answers, not just one!

- Do `[1, 3]` and `[2, 7]` merge to `[1, 2, 3, 7]`?

```
?- merge([1, 3], [2, 7], [1, 2, 3, 7]).
true.
```

Weaknesses and strengths

- Need to be careful about forming infinite loops
- Excels in database information retrieval; e.g. query systems
- Excels in domain-specific tasks:
  - Yacc is a declarative language: “here is a grammar for a language” not “here is a parsing procedure.”
  - Bddbddb uses Datalog to make queries to Java bytecodes: “does this field point to values returned by this method?”

Pattern matching examples

- Matches any three-element list that begins with an **`a`** and ends with a **`c`**:
  - `[a, X, c]`
- Matches any three-element list that begins with **`job`**, a second element of anything, and the third is a list of two elements that begins with **`computer`**:
  - `[job, X, [computer, Y]]`
- Matches any three-element list that begins with an **`a`** and whose second and third element can be anything as long as they are the same as each other:
  - `[a, X, X]`

Pattern matching algorithm

- We can keep track of variable assignments (in order to maintain consistency) by using frames.

```
(match (?x ?y ?y ?x) (a b b a) frame)    frame: x = a
result: ?y = b
```

```
(match (?x ?y ?y ?x) (a b b a) frame)    frame: y = a
result: fail
```

- Match takes in: a pattern, a datum, and a frame.

```
(match pattern-with-question-marks  datum  frame)
```

Pattern matching algorithm

```
(define (pattern-match pat dat frame)
  (cond [(eq? frame 'failed) 'failed]
        [(equal? pat dat) frame]
        [(var? pat) (extend-if-consistent pat dat frame)]
        [(and (pair? pat) (pair? dat))
         (pattern-match (cdr pat)
                        (cdr dat)
                        (pattern-match (car pat)
                                       (car dat)
                                       frame))]
        [else 'failed]))

(define (extend-if-consistent var dat frame)
  (let ([binding (binding-in-frame var frame)])
    (if binding
        (pattern-match (binding-value binding) dat frame)
        (extend var dat frame))))

(define (var? exp)
  (tagged-list? exp '?))
```

Unification

- Unification is a generalization of pattern matching, where the datum can have variables too.

```
[X, a, Y] = [Y, Z, a]
X = a,
Y = a,
Z = a.
```

```
[X, X] = [[a, Y, c], [a, b, Z]]
X = [a, b, c],
Y = b,
Z = c.
```

```
[X, a] = [[b, Y], Z]
X = [b, Y],
Z = a.
```

```
[X, Y, a] = [X, b, Y]
fail
```

Unification algorithm

```
(define (pattern-match p1 p2 frame)
  (cond [(eq? frame 'failed) 'failed]
        [(equal? p1 p2) frame]
        [(var? p1) (extend-if-possible p1 p2 frame)]
        [(var? p2) (extend-if-possible p2 p1 frame)]
        [(and (pair? p1) (pair? p2))
         (unify-match (cdr p1) (cdr p2)
           (unify-match (car p1) (car p2) frame))]
        [else 'failed]))

(define (extend-if-possible var val frame)
  (let ([binding (binding-in-frame var frame)])
    (cond [binding
           (unify-match (binding-value binding) val frame)]
      [(var? val)
       (let ([binding (binding-in-frame val frame)])
         (if binding
             (unify-match var (binding-value binding) frame)
             (extend var val frame))))
      [(depends-on? val var frame) 'failed]
      [else (extend var val frame)])
```

The Lambda Calculus

- Developed in 1930's by Alonzo Church and his students
- Studied in logic and computer science
- Church Thesis: "Effectively calculable functions from positive integers to positive integers are just those definable in the lambda calculus."
- Alan Turing around the same time developed Turing Machines, shown to be equivalent to lambda calculus.

The Lambda Calculus

- The only values are **functions that take a single argument**
- Normal order semantics

  Expression → *Id*

  Expression → `(` `lambda` *Id* `.` *Expression* `)`

  Expression → `(` *Expression Expression* `)`

- We'll abbreviate `lambda` as λ.
- We sometimes omit the parenthesis, in which case:
  - Application associates to the left:
    `x y` *means* `(x y)`
    `x y z` *means* `((x y) z)`
  - Abstraction extends to the right, as far as possible:
    `λx.x λy.x y` *means* `λx.(x (λy.x y)`
    `λx.x λy.x y z` *means* `λx.(x (λy.((x y) z)))`

The `identity` function

- Let's define:

1. The `identity` function:

2. A function that, given an argument `y`, discards it, and returns the `identity` function:

3. A function that, given a function `f`, invokes it on the `identity` function:

Simulating multiple arguments

- To define meaningful `cons`, `+`, `*` operations we'll need to simulate multiple arguments.
- Idea: To compute `a + b`, pass `a` to a function ⊕ that returns a function that, when applied to `b`, returns `a + b`.
  - That is: `(+ a b)` = `((⊕ a) b)`
  - What does `(⊕ a)` return? A function that adds `a` to its argument!
  - E.g., `(⊕ 1)` is the "increment by 1" function; also known as the `successor` function, or `succ`

Currying

- This simulating-multiple-arguments technique is called ***currying***, named after Haskell Curry.
- Let's use currying to define **`cons`**:

```
cons = (λa. (λb.
          (λselector. ((selector a) b))))
     = λa. λb. λselector. (selector a b)

car = λp.(p (λa. λb. a))

cdr = λp.(p (λa. λb. b))
```

- The conspiracy!

Make your own Boolean

- Key Idea: We want to ***encode the behavior of values***.
- E.g., to define Booleans:
  - Q: “What can we *do* with a Boolean?”
  - A: “Make a binary choice”
  - Q: “How can you view this as a function?”
  - A: “A Boolean is a function that takes two choices, returns one”

```
true  = λx. λy. x
false = λx. λy. y
```

```
true a b → a
false a b → b
```

Complete the conspiracy

- We don't have to let the "low-level implementation" of Booleans to be exposed to everyone (e.g., everyone knows which of the true and false alternatives is first).
- Instead, wrap this detail inside of "if," which acts like the Scheme `if` because *λ-calculus has normal order evaluation:*

```
true = λx. λy. x
false = λx. λy. y
if = λp. λthen. λelse. (p then else)
```

Boolean logic

- Let's define:

1. Not: Takes in a single argument b, assumed to be a Boolean, and returns the negation of b:

2. Or: Takes in two Booleans, b and c, and returns true if b or c are true:

3. And: Takes in two Booleans, b and c, and returns true if b and c are true:

Church numerals

- Q: “What can we do with a natural number?”
- A: “Iterate a number of times over some function”
- So, a number n can be a function that takes in a function, call it s, and applies that function to a base value, call it z, n times.
- E.g.:

```
0 = λs. λz. z                 /* s applied three times */
1 = λs. λz. (s z)             /* s applied once */
2 = λs. λz. (s (s z))         /* s applied twice */
3 = λs. λz. (s (s (s z)))     /* s applied three times */
```

Church numerals

- Numbers are functions that make a promise: “If you give me any zero function z and any successor function s, then I’ll apply s to z the number of times as the number that I represent.”
- Successor, given n, returns n+1:

```
succ = λn. (λs. λz. (s (n s z)))
```

- “Let’s apply s to z n times, and then apply s once more.”

Addition and multiplication

```
plus = λn. λm.(λs. λz. (m s (n s z)))
plus = λn. λm.(n succ m)
```

- Apply `s` to `z`, n times
- Then, apply `s` to that, m times
- Thus, we could let `succ = (plus 1)`

- Apply "something" to zero, m times.

```
mult = λn. λm.(m                    0))
```

- Now, `(expt n m)` = n^m

```
expt = λn. λm.
```

Subtraction

- Defined in terms of a predecessor function.
  - **pred(n)** = n - 1 if $n > 0$
  - **pred(n)** = 0 if $n = 0$
- We won't worry about negative numbers, instead, $n - m = 0$ if $m \geq n$.

```
subtract = λn. λm.
    (m pred n)
```

Predecessor

- This one is much harder, and stumped logicians for a while.
- Initialize a, b to 0.
- Repeatedly transform:
  - $a \leftarrow a + 1$
  - $b \leftarrow a$

A	B
0	*0*
1	*0*
2	*1*
3	*2*
4	*3*

- After the transform is applied n times, b is the predecessor of n!
- We don't even need to loop:

```
nextpair = λp.
      cons (succ (car p)) (car p)

pred = λn.
      cdr (n nextpair (cons zero zero))
```