Class: CSc 335

Date: Mar 23, 2023 (Thursday)

This is a function - wouldn't it be nice if we had an operator derivative which for suitable functions f, returned the function f'?

$$f'(x) = \lim_{\Delta \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

We can (almost) do that in scheme - we can design deriv to return the difference quotient

$$lambda\, f = \frac{f(x+dx) - f(x)}{dx}$$

for any choice of dx

Here it is:

Disclaimer: horrible numberics

```
(define (deriv f)
  (let ((dx 0.0000000001))
    (lambda (x) (/ (- (f (+ x dx))
                      (f x))
                   dx))))

((deriv (lambda (x) (* x x x))) 5) ;; 75.000059496233
```

To evaluate the (pseudo) scheme code in TLS, you'll need to use the *quote operator*. Quote serves to block evaluation.

In TLS you may see something like

```
(list dog cat)
```

neither `dog` nor `cat` is the name of a value.

What TLS has in mind: `dog` and `cat` are symbols, not names. Quote is the mechanism which allows LISP languages to treat identifies (and indeed anything else) as symbols - Quote is abbreviated by `'`

```
(quote dog) = dog => 'dog = dog
(quote cat) = cat => 'cat = cat
(quote (+ 3 4)) = (+ 3 4) => '(+ 3 4) = (+ 3 4)
```

S-expressions

The BNF for s-expressions can be used generatively or analytically

NOTE: All post-conditions in the upcoming quiz will be in terms of BNF.

```
(define (s-exp? l)
  (cond ((null? l) #t)
        ((eq? l 'a) #t)
        ((eq? l 'b) #t)
        (()) ;; some recursive processing of list...
        (else #f)))

;; we have to worry that about how #f could be returned
```

How to critique a proposed BNF definition of some class?

There are 2 criteria:

1. Soundness
    - Is it the case that everything generated belongs to the class we seek to define?
    - In this class, for the time being:
    - careful def is coming soon
        - For now: a pair is something with first and second elements

```
(define (atom? x)
    (and (not (null? x))
    (not (pair? x))))
```

2. Completeness
    - Is it the code that the def. denerates everything in the class?
    - concerned about completeness because this def does not permit lists of (non-atoms??) lists

---

What is Structural Induction?

- To do it, we need a structure i.e. we need an inductive definition.
- We said: s-exp over atoms a, b is the least class containing a and b and () empty list, which is closed under the operation of forming finite list.
    - i.e. if $s_1, s_2, ..., s_f$ are s-exps, this so is $(s_1, s_2, ..., s_f)$
- Similarily - the class P_v of propositions over $V = \{x, y\}$ is the least class containing T, F are X, Y which is closed under
    - AND : if P, Q are propositions then so is $P\ \ AND\ Q$
    - OR : if P, Q are propositions then so is $P\ \ OR\ Q$
    - NOT : if P, Q are propositions then so is $P\ \ NOT\ \ Q$
    - $=>$: if P, Q are propositions then so is $P\ \ => \ \ Q$

How can we leverage these inductive defs to carry out a structural induction?

For a structural indction, one requires the notion of proper component once we have this, the IH amounts to the assmption that the desired conclusion is true for all proper components.

For example, we can use this technique to show that any proposition written with `AND, OR, NOT, =>` is logically equivalent to one written using only the operations `AND, NOT`.

BASIS STEP

- Vacuously true for X, Y, T, F

Induction Hypothesis: Assume true for proper components of the prop currently under consideration

Induction Step:

1. P has the form $R \vee S$
    - By the IH, we have $R' \equiv R$ and $S' \equiv S$ where the only ops in R' and S' are `AND, NOT` so we're done if we can remove the final $\vee$ from $R' \vee S'$. But this can be done with DeMorgan's Law.
    - DeMorgan's Law:
    - $x \vee y \equiv \neg(\neg x \wedge \neg y)$
    - So $R' \vee S' \equiv \neg(\neg R' \wedge \neg S')$
    - Recall $P \implies Q \equiv \neg P \vee Q$