Recitation 3: Scheme

CS 105

Goals

1. Design functions that consume lists.
2. Prove properties of functions that consume lists.

Part I: Designing functions that consume lists

In this recitation, you will use lists to represent mathematical sets. Specifically, we represent a set as a list with no repeating elements; the functions that create and manipulate these "sets" must maintain the key property that no element is repeated (this is an example of what's called a *representation invariant*).

All S-expressions mentioned here are "ordinary", which can be defined inductively as either an atom or a (possibly empty) list of S-expressions.

For this first part, you will complete some design steps for two functions:

- `(member? x s)`, which tells if S-expression `x` is in set `s`
- `(add-elem x s)`, which returns the set union of set `s` and a singleton set containing `x`

Start with the first 6 steps for `member?`:

1. *Forms of data.* Both arguments of `member?` have observable forms of data; list the forms each argument can take:

2. *Example inputs.* Provide an example input for each form of data identified in the previous step. (*Hint*: The sooner you are comfortable writing list literals using `'`, the better. But don't write `'` inside another `'`, as in `'('a)` and `'((1 'b))`. Instead, write `'(a)` and `'((1 b))`.)

3. *Function's name* is given as `member?`.

4. *Function's contract* is given above.
5. *Example results.* Given below are example unit tests for `member?` using `check-assert`. You ultimately want, if possible, one "member" and one "not member" test for each form of the set argument `s`.

```
(check-assert (not (member? 'a '())))
(check-assert      (member? 'a '(a b c)))
(check-assert (not (member? 'a '(1 2 3 4))))
```

6. Finally, generalize the unit tests from step 5 into *algebraic laws.*

For `add-elem`, the first four steps of the design process are essentially done, e.g., the same example inputs work here. We'll do steps 5 and 6:

5. *Example results.* Given below are example unit tests for `add-elem`, this time using `check-expect`.

```
(check-expect (add-elem 'a '())      '(a))
(check-expect (add-elem 'a '(a b c)) '(a b c))
(check-expect (add-elem 'a '(1 2 3 4)) '(1 2 3 4 a))
```

6. Generalize the unit tests into *algebraic laws* for `add-elem`.

Part II: Proof of a property (induction)

If we add element `x` to a set, then `x` should be a member of the new set. We can express this expectation as a *property*, which is a form of algebraic law useful for testing. Here is the property, which we will call the `member-add` property:

```
(member? x (add-elem x xs)) == #t
```

Since `x` and `xs` are not otherwise specified, this property is meant to be true for *any* value of `x` and `xs`, provided only that these values respect the contracts for `member?` and `add-elem`. We'll prove this property by structural induction on `xs` with one case for the empty list and two non-empty lists, one that starts with `x` and another that starts with $\texttt{z} \neq \texttt{x}$.

Although you can prove a property like this one by looking at the source code for the named functions, there's a simpler, easier way: by using the algebraic laws that designed the source code! Here they are, with names:

```
(member? x '())          = #f      ; member?-empty law
(member? x (cons x xs))  = #t      ; member?-same law
(member? x (cons y xs))  = (member? x xs), when x != y
                                   ; member?-different law

(add-elem x '())         = (cons x '())  ; add-empty law
(add-elem x (cons x xs)) = (cons x xs)   ; add-same law
(add-elem x (cons y xs)) = (cons y (add-elem x xs)), when x != y
                                         ; add-different law
```

Given just these laws (i.e., you don't need to see the code for `member?` or `add-elem`), use structural induction on `xs` to prove the property

```
(member? x (add-elem x xs)) == #t
```

If you can't quite remember how to do calcuational proofs, don't worry- this is where you'll get practice. See Appendix: Getting Started with a Calcuational Proof for a memory bolster.

Key step in the proof: The primary idea of these kinds of proofs is "equational reasoning": use the algebraic laws and the form of `xs` to build a chain of equalities connecting the left side of the property to the right. But there is a key additional step, which is the application of the induction hypothesis. That step works like this: when `xs` takes the form `(cons y ys)`, the induction hypothesis states that

```
(member? x (add-elem x ys)) = #t
```

There is no need to prove this equality: it is given as the induction hypothesis, and either side of this equality can be substituted for the other.

Write the proof for Part II here.

Appendix: Getting Started with a Calcuational Proof

You've seen calculational proofs before in lecture, but that may have been your first time experiencing the new format. As a reminder, the way to write a calculational proof is as follows:

1. Gather the algebraic laws relevant to the data (you are provided those here)
2. Prove the base case holds by subsequent application of the laws to rewrite terms

a. This looks like:

term

= {name-of-law}

new-term

= {...}

As in most proofs, it is helpful to know the destination. Writing backwards from there can be extremely helpful if you're stuck on the forward journey.

Remember, you hope to arrive at a term that concretely proves what you're trying to say. This likely will be the right-hand side of some equality claim you've made.

3. Form an inductive hypothesis about generalized data (again, given here)
4. Prove the claim holds for the general case using the inductive hypothesis. Again, this looks like:

term

= {name-of-law}

new-term

= {...}

Crucially, one of the steps in `{}` will be "Application of the induction hypothesis" or "by the induction hypothesis."

Finally, here are the lecture slides on a specific case for `member` and a general case for `length` to help recall formatting.

Here is the start of a proof on specific data using `member`, with its definition removed (no cheating!):

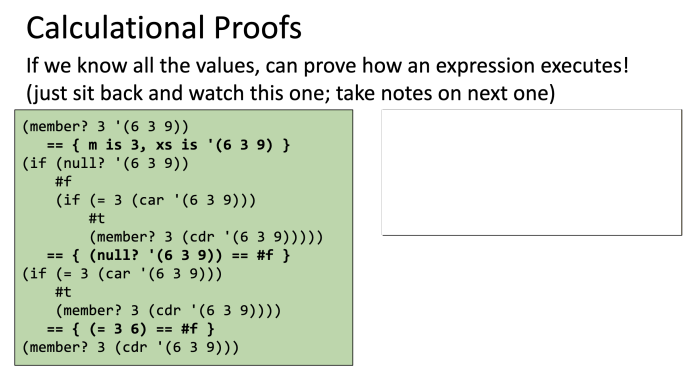

Figure 1: member-specific

Here is the base-case step of a general proof involving `length`:

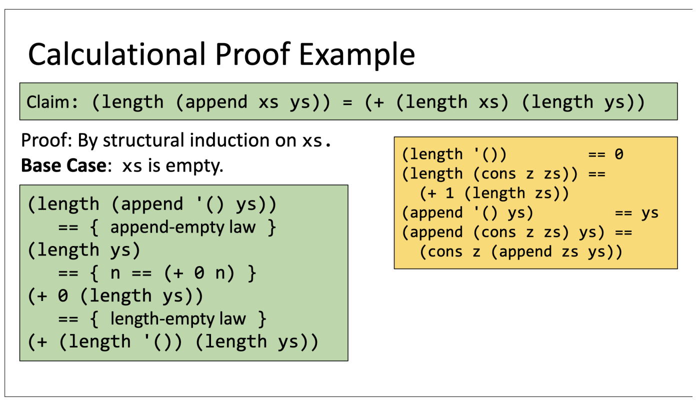

Figure 2: length-base

Finally, here is the inductive step of a general proof involving `length`. Note the step that includes application of the inductive hypothesis.

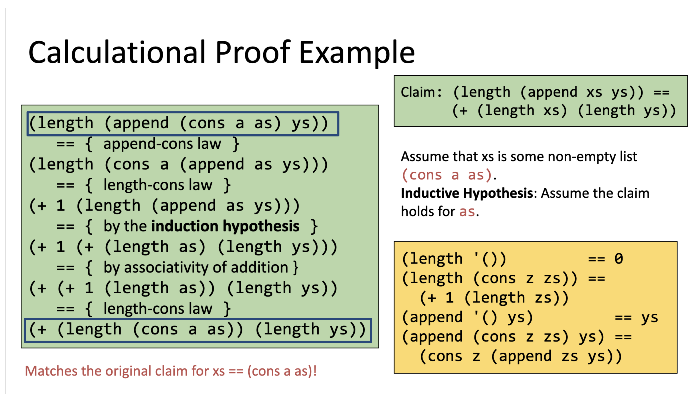

Figure 3: length-inductive