Higher-Order Functions

CS 105 Assignment

Due Tuesday, October 4, 2022 at 11:59PM

Contents

This assignment is all individual work. There is **no pair programming**.

Overview

Higher-order functions are a cornerstone of functional programming. They are found in all of the web/scripting languages, including JavaScript, Python, Perl, and Lua. This assignment will help you incorporate first-class and higher-order functions into your programming practice. You will use existing higher-order functions, define higher-order functions that consume functions, and define higher-order functions that return functions. The assignment builds on what you've already done, and it adds new ideas and techniques that are described in sections 2.7, 2.8, and 2.9 of *Build, Prove, and Compare*.

Setup

The executable μScheme interpreter is in `/comp/105/bin/uscheme`; if you are set up with `use comp105`, you should be able to run `uscheme` as a command. The interpreter accepts a `-q` ("quiet") option, which turns off prompting. Your homework will be graded using `uscheme`. When using the interpreter interactively, you may find it helpful to use `ledit`, as in the command

```
ledit uscheme
```

We don't give you a template—by this time, you know how to identify solutions and where to put contracts, algebraic laws, and tests.

Dire Warnings

The μScheme programs you submit must not use any imperative features. **Banish `set`, `while`, `print`, `println`, `printu`, and `begin` from your vocabulary! If you break this rule for any exercise, you get No Credit for that exercise.** You may find it useful to use `begin` and `println` while debugging, but they must not appear in any code you submit. As a substitute for assignment, use `let` or `let*`.

No code may compute the length of a list.

Except as noted below, do not define helper functions at top level. Instead, use `let` or `letrec` to define helper functions. When you do use `let` to define inner helper functions, avoid passing as parameters values that are already available in the environment. (An example of what to avoid appears under "Avoid common mistakes" below.)

Your solutions must be valid μScheme; in particular, they must pass the following test:

```
/comp/105/bin/uscheme -q < myfilename > /dev/null
```

without any error messages or unit-test failures. If your file produces error messages, we won't test your solution and you will earn No Credit for functional correctness. (You can still earn credit for structure and organization). If your file includes failing unit tests, you might possibly get some credit for functional correctness, but we cannot guarantee it.

Every function should be accompanied by a short contract and by unit tests. If the function does case analysis, it must also be accompanied by algebraic laws. Submissions without algebraic laws will earn **No Credit**.

We will evaluate functional correctness by testing your code extensively. **Because this testing is automatic, each function must be named be exactly as described in each question. Misnamed functions earn No Credit.**

Reading comprehension

Before starting the programming problems, answer the reading-comprehension questions for this module, which you will find online. It is also OK to alternate between reading-comprehension questions and related homework questions.

Programming and Proof

Overview

For this assignment, you will do Exercises **28 (b, e, f)**, **29 (a, c)**, **30**, and **38**, from pages 186 to 189 of *Build, Prove, and Compare*, plus the exercises **F**, **M**, **O**, and **V** below.

A summary of μScheme's initial basis can be found on page~99. While you're working on this homework, *keep it handy*.

Each top-level function you define must be accompanied by a contract and unit tests. Each *named* internal function written with `lambda` should be accompanied by a contract, but internal functions cannot be unit-tested. (Anonymous `lambda` functions need not have contracts.) Algebraic laws are required only where noted below; each problem is accompanied by a **Laws** section, which says what is needed in the way of algebraic laws.

Book problems

---

28. *Higher-order functions and nonempty lists*. Do exercise~28 on page~186 of *Build, Prove, and Compare*, parts (b), (e), and (f). These functions accept *nonempty* lists and produce scalar results. Code accordingly. **You must *not* use recursion—solutions using recursion will receive No Credit.**

Because you are not defining recursive functions, you need not write any algebraic laws.

For this problem only, you may define *one* helper function at top level.

Related reading: For material on higher-order functions, see section~2.8, which starts on page~129.

Laws: These functions must not be recursive, should not do any case analysis,[1] and do not return functions. Therefore, no algebraic laws are needed.

---

29. *Higher-order functions for list functions*. Do exercise~29 on page~186 of *Build, Prove, and Compare*, parts (a) and (c): use higher-order functions to implement two standard recursive functions without recursion. **You must *not* use recursion—solutions using recursion will receive No Credit.**

Because you are not defining recursive functions, you need not write any algebraic laws.

Related reading: For material on higher-order list functions, see section~2.8, which starts on page~129. For material on `curry`, see section~2.7.2, which starts on page~127.

Laws: These functions must not be recursive, should not do any case analysis,[2] and do not return functions. Therefore, no algebraic laws are needed.

---

[1] Case analysis may be happening, but on this problem, it will be happening inside functions like `map` and `foldr`, not in any code that you write.

[2] Case analysis may be happening, but on this problem, it will be happening inside functions like `map` and `foldr`, not in any code that you write.

30. *Folds for higher-order functions*. Do exercise~30 on page~186: implement four standard higher-order functions using only folds. **You must *not* use recursion—solutions using recursion will receive No Credit.** (It is OK if recursion happens in functions you *call*, like `foldl` and `foldr`. But it must not happen in functions you *write*.)

Because you are not defining recursive functions, you need not write any algebraic laws.

For this problem, you get full credit if your implementations return correct results. You get *extra* credit[3] if you can duplicate the behavior of `exists?` and `all?` exactly. To earn the extra credit, it must be impossible for an adversary to write a μScheme program that produces different output with your version than with a standard version. However, the adversary is not permitted to change the names in the initial basis.

Related reading: Examples of `foldl` and `foldr` are in sections 2.8.1 and 2.8.2 starting on page~129. You may also find it helpful to study the implementations of `foldl` and `foldr` in section~2.8.3, which starts on page~131; the implementations are at the end. Information on `lambda` can be found in section 2.7, on pages 122 to 127.

Laws: These functions must not be recursive, should not begin with case analysis, and do not return functions. Therefore, no algebraic laws are needed.

---

38. *Functions as values*. Do exercise~38 on page~189 of *Build, Prove, and Compare*. **You cannot represent these sets using lists.** If any part of your code to construct or to interrogate a set uses `cons`, `car`, `cdr`, or `null?`, you are doing the problem wrong.

Do all four parts:

- Parts (a) and (b) require no special instructions.
- In part (c), your `add-element` function must take two parameters: the element to be added as the first parameter and the set as the second parameter.

  When you code part (c), compare values for equality using the `equal?` function.

  To help you design part (c), put comments in your source code that complete the right-hand sides of the following *properties*:

  ```
  (member? x (add-element x s)) == ...
  (member? x (add-element y s)) == ..., where (not (equal? y x))
  (member? x (union s1 s2))     == ...
  (member? x (inter s1 s2))     == ...
  (member? x (diff  s1 s2))     == ...
  ```

  The properties are not quite algorithmic, but they should help anyway.
- In part (d), when you code the third approach to polymorphism, write a function `set-ops-from` which places your set functions in a record. To define record functions, use the syntactic sugar described in the book in Section 2.13.6 on page~169. In particular, be sure your code includes this record definition:

  ```
  (record set-ops [emptyset member? add-element union inter diff])
  ```

[3] In your README, please identify this credit as EXACT-EXISTS.

Code your solution to part (d) as a function `set-ops-from`, which will accept one argument (an equality predicate) and will return a record created by calling `make-set-ops`. Your function might look like this:

```
(define set-ops-from (eq?)
  (let ([emptyset    ...]
        [member?     ...]
        [add-element ...]
        [union       ...]
        [inter       ...]
        [diff        ...])
    (make-set-ops emptyset member? add-element union inter diff)))
```

Fill in each `...` as appropriate. Each implementation is like the one given in the book or the one you wrote in part (c), except instead of using the predefined `equal?`, it uses the parameter `eq?`—that is what is meant by "the third approach to polymorphism."

No additional laws are needed for part (d).

To help you get part (d) right, we recommend that you use these unit tests:

```
(check-assert (function? set-ops-from))
(check-assert (set-ops? (set-ops-from =)))
```

And to write your own unit tests for the functions in part (d), you may use these definitions:

```
(val atom-set-ops     (set-ops-from =))
(val atom-emptyset    (set-ops-emptyset    atom-set-ops))
(val atom-member?     (set-ops-member?     atom-set-ops))
(val atom-add-element (set-ops-add-element atom-set-ops))
(val atom-union       (set-ops-union       atom-set-ops))
(val atom-inter       (set-ops-inter       atom-set-ops))
(val atom-diff        (set-ops-diff        atom-set-ops))
```

Hint: The recitation for this unit includes an "arrays as functions" exercise. Revisit it.

Related reading: For functions as values, see the examples of `lambda` in the first part of section 2.7 on page~122, and also the array exercise from recitation. For function composition and currying, see section 2.7.2. For polymorphism, see section~2.9, which starts on page~133.

Laws: Complete the right-hand sides of the properties listed above. These properties say what happens when `member?` is applied to any set created with any of the other functions. No other laws are needed.

A function that returns a function

F. The third lesson in program design ("Higher-order functions") mentions a higher-order function `flip`, which can convert `<` into `>`, among other tricks. Using your algebraic law or laws from the comprehension questions, define `flip`. Don't forget unit tests.

Related reading: *Seven Lessons in Program Design*, lesson 3.

Laws: Use your law or laws from the comprehension questions.

Calculational reasoning about functions

M. *Reasoning about higher-order functions*. Using the calculational techniques from Section 2.5.7, which starts on page~116, prove that

```
(o ((curry map) f) ((curry map) g)) == ((curry map) (o f g))
```

To prove two functions equal, prove that when applied to equal arguments, they return equal results.

Related reading: Section 2.5.7. The definitions of composition and currying in section 2.7.2. Example uses of `map` in section 2.8.1. The definition of `map` in section 2.8.3.

Laws: In this problem you don't write new laws; you reuse existing ones. You may use any law in the Basic Laws handout, which includes laws for `o`, `curry`, and `map`. (If it simplifies your proof, you may also introduce new laws, provided that you prove each new law is valid.)

Ordered lists

O. *Ordered lists*. Like natural numbers, the forms of a list can be viewed in different ways. In almost all functions, we examine just two ways a list can be formed: `'()` and `cons`. But in some functions, we need a more refined view. Here is a problem that requires us to divide a list of values into *three* forms.

Define a function `ordered-by?` that takes one argument—a comparison function that represents a transitive relation—and returns a predicate that tells if a list of values is totally ordered by that relation. Assuming the comparison function is called `precedes?`, here is an inductive definition of a list that is ordered by `precedes?`:

- The empty list of values is ordered by `precedes?`.
- A singleton list containing one value is ordered by `precedes?`.
- A list of values in the form `(cons x (cons y zs))` is ordered by `precedes?` if the following properties hold:
  - `x` is related to `y`, which is to say `(precedes? x y)`.
  - List `(cons y zs)` is ordered by `precedes?`.

Here are some examples. Note the parentheses surrounding the calls to `ordered-by?`.

```
-> ((ordered-by? <) '(1 2 3))
#t
-> ((ordered-by? <=) '(1 2 3))
#t
-> ((ordered-by? <) '(3 2 1))
#f
-> ((ordered-by? >=) '(3 2 1))
#t
-> ((ordered-by? >=) '(3 3 3))
#t
-> ((ordered-by? =) '(3 3 3))
#t
```

Hints:

- The entire 9-step software-design process applies, and it starts with the *three* forms of data in the definition of "list ordered by" above.
- For the code itself, we recommend `letrec`. You can emulate the example of the polymorphic sort function on page~137.
- We recommend that your submission include the following unit tests, which help ensure that your function has the correct name and takes the expected number of parameters.

```
(check-assert (function? ordered-by?))
(check-assert (function? (ordered-by? <)))
(check-error (ordered-by? < '(1 2 3)))
```

Related reading: Section 2.9, which starts on page~133. Especially the polymorphic sort in section 2.9.2—the `lt?` parameter to that function is an example of a transitive relation.

Laws: Write algebraic laws for `ordered-by?`, including at least one law for each of the *three* forms of data used in the definition of "list ordered by" above.

A domain-specific language for input validation

In this part of the homework, you will use higher-order functions to develop an *embedded domain-specific language* for validating web-form submissions.[4]

The web version of this homework displays an example web form for CS 105 regrades. In a real form, clicking on the `Submit` button sends data to the server, and that data has to be validated. For example, if the form asks for a grade to be reviewed, it must specify an assignment. (To try our validator for yourself, go to the course regrade form at https://www.cs.tufts.edu/comp/105/regrade, and click the Submit button without filling in the form.) Our validator checks the form's input using higher-order functions, and for problem **V** below, you will define similar functions.

Representation of form data and validation results

When form data reaches the server, it is turned into an association list, which we call a *response*. In the response, each key names a field of the form; the key is represented by a symbol. The key is bound to the value the user responded with. If the user did not supply a value, the key is bound to the value `#f`. Here's an example response, representing a student who wants something regraded because they accidentally submitted the wrong PDF:

```
(val sample-response
  '([why badsubmit]
    [badsubmit_asst scheme]
    [info (I accidentally submitted the opsem PDF again.)]
    [badgrade_asst ...]
    [problem #f]))
```

But what if they say they submitted the wrong PDF, but they forgot to say what assignment? The validator needs to identify such *faults*. For our purposes, a fault is simply a μScheme symbol that identifies a bad input. And a *validator* is a function that takes a response and returns a list of faults (without duplicates):

input and output of a single validator

[4] It's actually just a library, but a library like this is called a "language" because of the way functions compose. The composition of library functions resembles the syntactic composition of expressions in an actual programming language.

A language of validators

The key idea behind the validator functions—what makes them a "domain-specific *language*" and not just another API—is *composition*: you define functions that build validators from other validators. The functions are specified using these metavariables:

- F, a fault (a symbol)
- k, a key (also a symbol)
- v, a value
- V_1 and V_2, validators
- t, a *validator table*, which is an association list that maps values to validators

Here are the specifications:

- `(faults/none)` returns a validator that takes a response and always returns the empty list of faults, no matter what is in the response.

  Validator constructor `faults/none`

- `(faults/always` F`)` returns a validator that takes a response and always returns a singleton list containing the fault F, no matter what is in the response.

  Validator constructor `faults/always`

- `(faults/equal` k v`)` returns a validator that takes a response and if k is bound to v in the response, returns a singleton list containing k. If k is not bound to v in the response, the validator returns the empty list of faults.

  Validator constructor `faults/equal`

- `(faults/both` V_1 V_2`)` returns a new validator that combines checks from the two validators V_1 and V_2. The `faults/both` validator returns a single list of faults that is formed by taking the union of the faults from V_1 with the faults from V_2 (with no duplicates).

  Validator constructor `faults/both`

- `(faults/switch` k t`)` returns a validator that validates the response using a validator selected from validator table t: the value of k in the response is a key used to find a validator in t.

  Validator constructor `faults/switch`

  The use case for `faults/switch` is to create a validator that uses some of the input data to figure out how to validate the rest, and the validator table is like the list of cases in a C `switch` statement: each case is labeled with a value, and the action to be performed in that case is itself a validator. There's an example below.

  The contract of `faults/switch` is hard to write in informal English: "Key k determines which field of the response is used to make a decision. When the validator is given a response, the value v associated with key k *in the response* is looked up in the validator table, and the resulting validator is used to find faults in the response." We're actually better off documenting the function with an algebraic law. In a language like Lua, Python, or JavaScript, where table lookup is primitive, I might write the essential part of an algebraic law like this:

  ```
  faults_switch(k, t)(response) == t[response[k]](response)
  ```

  When you write your algebraic laws, you will notate a similar computation in µScheme.

(An example appears on the next page.)

An example validator

To illustrate the use of the validator functions, here is some code for the regrade validator. It will help to know that it uses these faults:

- `nobutton` if no radio button was selected
- `problem` if "review my grade" was selected but the "problem" field was blank
- `badsubmit_asst` if "I submitted the wrong PDF" was selected but the "assignment" dropdown was left at "`...`"
- `recitation` if "I attended this recitation" was selected but the date or leaders were left blank

If the regrade validator were written in μScheme, it might look like this:

```
(val regrade-validator  ;; example for the regrade form
  (faults/switch 'why
    (bind         'photo
                  (faults/none)
      (bind       'badsubmit
                   (faults/both (faults/equal 'badsubmit_asst '...)
                                (faults/equal 'info #f))
        (bind     'badgrade
                  (faults/both
                      (faults/equal 'badgrade_asst '...)
                      (faults/both
                         (faults/equal 'info #f)
                         (faults/equal 'problem #f)))
          (bind   'recitation
                  (faults/both
                      (faults/equal 'date #f)
                      (faults/equal 'leaders #f))

            (bind '#f
                  (faults/always 'nobutton)
                  '()))))))))
```

For example, if I submit a photo for regrade, `faults/switch` figures out that no more validation is required. But if I select "review my grade" (`badgrade`), `faults/switch` figures out that we need an assignment (`badgrade_asst`), an explanation (`info`) and a problem number (`problem`). If I leave all these fields blank, the validator will return a fault list containing symbols `problem`, `badgrade_asst`, and `info`, like this one:

```
'(problem badgrade_asst info)
```

If I leave only one or two of these fields blank, the validator will return a fault list identifying those fields.

You can use this validator to test your functions with some integration tests.

At last, your assigned problem

V. Implement validation functions `faults/none`, `faults/always`, `faults/equal`, `faults/both`, and `faults/switch`. As usual, each function must be accompanied by algebraic laws and unit tests. The five functions in the solutions total less than 20 lines of µScheme.

Additional expectations and other notes:

- Every validator must treat the response as an abstraction. A validator must never interrogate a response about its form of data; the validator should restrict itself to function `find` and possibly function `bound-in?`:

  ```
  (define bound-in? (key pairs)
    (if (null? pairs)
        #f
        (|| (= key (alist-first-key pairs))
            (bound-in? key (cdr pairs)))))
  ```

- Most of your algebraic laws, like the algebraic laws for `curry` and `flip`, will need to specify what happens when the result of calling a function is itself applied. If written well, your laws will be clearer and easier to follow than the informal descriptions above. Which is one reason why we write them.

- Function `faults/none` is not itself a validator; it is a function that is called with no arguments and *returns* a validator. This design choice makes the design seem more regular: to get a validator, you *always* call a function.

- The implementation of `faults/switch` is simplest if key k is bound in the response to a value v that is bound in the validator table. When you write your algebraic laws, make this simplifying assumption. And when you write your code, you may simply assume that k is bound in the response. But if v is *unbound* in the validator table, the validator must halt with a checked run-time error. (Any error will do.)

If you like you can start with the template below, which provides a couple of algebraic laws, plus local bindings for `faults/both`. The local bindings will enable your fault code to avoid conflicts with the set code in exercise~38.

```
(val the-fault list1) ; build a singleton fault set
(val no-faults '())   ; an empty fault set

; ((faults/none) response) == no-faults
(define faults/none ()
    ...)

; ((faults/always f) response) == (the-fault f)
(define faults/always (f)
    ...)

; ... law or laws for faults/equal ...
(define faults/equal (key value)
    ...)

; ... law or laws for faults/both ...
(val faults/both
    (let* ([member?  (lambda (x s) (exists? ((curry =) x) s))]
           [add-elem (lambda (x s) (if (member? x s) s (cons x s)))]
           [union    (lambda (faults1 faults2)
```

```
                        (foldr add-elem faults2 faults1))])
    ...))

; ... law or laws for faults/switch ...
(define faults/switch (key validators)
    ...)
```

Related reading:

- For a review of association lists and `find`, consult section~2.3.8, which starts on page~107.
- All the functions in this problem are "function factories": each one takes in some arguments and returns a function. The main function factories in the book are `curry` and `o` ("compose"); to review them, study section~2.7.2, which starts on page~127.
- For `faults/both`, revisit the `combine` and `divvy` examples from the lecture on `lambda`. (These functions are intended to be called `conjoin` and `disjoin`.)

Extra credit

VX. For extra credit, answer the questions below.

(a) Which of the following equations are valid properties of the fault-validation functions?

- `(faults/both (faults/none)` V`)` $\equiv$ V?
- `(faults/both (faults/always` F`)` V`)` $\equiv$ `(faults/always` F`)`?

(b) For each of the equations in part (a),

- If the equation is a valid property, present a calculational proof using your laws from problem **V**.
- If the equation is not a valid property, present a counterexample. That is, present examples of V, F (if necessary), and a response such that, when applied to the response, the two validators produce different fault sets.

What and how to submit

You must submit three files:

- A `README` file containing
  - The names of the people with whom you collaborated
  - A list identifying which problems you solved
  - A note identifying any extra-credit work you did (Please use the words "extra credit" somewhere in your README.)
- A PDF file `semantics.pdf` containing the solutions to Exercise **M**. If you already know LaTeX, by all means use it. Otherwise, write your solution by hand and scan it. Do check with someone else who can confirm that your work is legible—if we cannot read your work, we cannot grade it.
- A file `solution.scm` containing the solutions to Exercises **28 (b, e, f)**, **29 (a, c)**, **30**, **38**, **F**, **O**, and **V**

  Each solution must be preceded by a comment that looks like something like this:

```
;;
;; Problem 28
;;
```

If you complete the extra-credit exercise **VX**, you must submit it separately:

- Optionally, a PDF file `extra.pdf` containing solutions to extra credit exercise **VX**

As soon as you have the files listed above, run `submit105-hofs` to submit a preliminary version of your work. Keep submitting until your work is complete; we grade only the last submission.

Avoid common mistakes

Listed below are some common mistakes, which we encourage you to avoid.

Passing unnecessary parameters. In this assignment, a very common mistake is to pass unnecessary parameters to a nested helper function. Here's a silly example:

```
(define sum-upto (n)
  (letrec ([sigma (lambda (m n) ;;; UGLY CODE
                    (if (> m n) 0 (+ m (sigma (+ m 1) n))))])
    (sigma 1 n)))
```

The problem here is that **the `n` parameter to `sigma` never changes**, and it is already available in the environment. To eliminate this kind of problem, don't pass the parameter:

```
(define sum-upto (n)
  (letrec ([sum-from (lambda (m) ;;; BETTER CODE
                    (if (> m n) 0 (+ m (sum-from (+ m 1)))))])
    (sum-from 1)))
```

The name of the internal function is different, but the only other things that are different is that the second formal parameter from the `lambda` is gone and as is the second actual parameter from the call sites. You can still use `n` in the body of `sum-from`; it's visible from the definition.

An especially good place to avoid this mistake is in your definition of `ordered-by?` in problem **O**.

Another common mistake is to fail to redefine predefined functions like `map` and `filter` in exercise~30. Yes, we really want you to provide new definitions that replace the existing functions, just as the exercise says.

How your work will be evaluated

Structure and organization

The criteria in the general coding rubric apply. As always, we emphasize **contracts** and **naming**. In particular, unless the contract is obvious from the name and from the names of the parameters, **an inner function defined with `lambda` and a `let` form needs a contract**. (An anonymous lambda that is returned from a function like `faults/both` does not need a contract—the behavior of that lambda is part of the contract of the function that returns it.)

There are a few new criteria related to `let`, `lambda`, and the use of basis functions. The short version is **use the functions in the initial basis; except when we specifically ask you to, don't redefine initial-basis functions.**

	Exemplary	Satisfactory	Must Improve
Structure	• Short problems are solved using simple anonymous `lambda` expressions, not named helper functions. • When possible, inner functions use the parameters and `let`-bound names of outer functions directly. • The initial basis of µScheme is used effectively.	• Most short problems are solved using anonymous lambdas, but there are some named helper functions. • An inner function is passed, as a parameter, the value of a parameter or `let`-bound variable of an outer function, which it could have accessed directly. • Functions in the initial basis, when used, are used correctly.	• Most short problems are solved using named helper functions; there aren't enough anonymous `lambda` expressions. • Functions in the initial basis are redefined in the submission.

Functional correctness

In addition to the usual testing, we want appropriate list operations to take constant time.

	Exemplary	Satisfactory	Must Improve
Correctness	• Your code passes every one of our stringent tests. • Testing shows that your code is of high quality in all respects.	• Testing reveals that your code demonstrates quality and significant learning, but some significant parts of the specification may have been overlooked or implemented incorrectly.	• Testing suggests evidence of effort, but the performance of your code under test falls short of what we believe is needed to foster success. • Testing reveals your work to be substantially incomplete, or shows serious deficiencies in meeting the problem specifications (**serious fault**). • Code cannot be tested because of loading errors, or no solutions were submitted (**No Credit**).
Performance	• Empty lists are distinguished from non-empty lists in constant time.		• Distinguishing an empty list from a non-empty list might take longer than constant time.

Proofs and inference rules

For your calculational proof, **use induction correctly** and **exploit the laws that are proved in the book**.

	Exemplary	Satisfactory	Must Improve
Proofs	• Proofs that involve predefined functions appeal to their definitions or to laws that are proved in the book. • Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction exactly where needed. • Each proof by induction states the induction hypothesis explicitly.	• Proofs involve predefined functions but do not appeal to their definitions or to laws that are proved in the book. • Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction, even if it may not always be needed. • A proof by induction does not state the induction hypothesis explicitly, but course staff can easily figure out what it is.	• A proof that involves an inductively defined structure, like a list or an S-expression, does **not** use structural induction, but structural induction is needed. • There is a proof by induction, but course staff cannot easily figure out what part of the proof is the induction hypothesis.