Higher-Order Functions

COMP 105 Assignment

Due Tuesday, February 20, 2018 at 11:59PM

Contents

Overview 1
Setup 2
Dire Warnings 2
Reading Comprehension (10 percent) 2
Programming and Proof (90 percent) 5
  Overview ................................................................. 5
  Book problems ......................................................... 5
  Relating imperative code to functional code ........................ 7
  Graph problems ....................................................... 8
  Calculational reasoning about functions ............................. 10
  Ordered lists .......................................................... 10

What and how to submit 11

Avoid common mistakes 12

How your work will be evaluated 12
  Structure and organization .......................................... 12
  Functional correctness ............................................... 13
  Proofs and inference rules .......................................... 14

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

Overview

The purpose of this assignment is to give you experience using first-class and higher-order functions so that you can incorporate them 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 set you up with 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, println, 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*.

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.

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.

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 (10 percent)

Answer these questions before starting the rest of the assignment. As usual, you can download the questions\(^1\).

1. The first step in this assignment is to learn the standard higher-order functions on lists, which you will use a lot. Suppose you need a list, or a Boolean, or a function—what can you call?

Review Sections 2.7.2, 2.8.1, and 2.8.2. Now consider each of the following functions:

```
map filter exists? all? curry uncurry foldl foldr
```

\(^1\)/cqs.hofs.txt
Put each function into exactly one of the following four categories:

(B) Always returns a Boolean
(F) Always returns a function
(L) Always returns a list
(A) Can return anything (including a Boolean, a function, or a list)

After each function, write (B), (F), (L), or (A):

map
filter
exists?
all?
curry
uncurry
foldl
foldr

2. Here are the same functions again:

map filter exists? all? curry uncurry foldl foldr

For each function, say which of the following five categories best describes it. Pick the most specific category (e.g., (S) is more specific than (L) or (M), and all of these are more specific than (?)).

(S) Takes a list and a function and always returns a list of the same size
(L) Takes a list and a function and always returns a list of at least the same size
(M) Takes a list and a function and always returns a list of at most the same size
(?) Might return a list
(V) Never returns a list

After each function, write (S), (L), (M), (?), or (V):

map
filter
exists?
all?
curry
uncurry
3. Here are the same functions again:

```
foldl
foldr
```

Put each function into exactly one of the following categories. Always pick the most specific category (e.g. (F2) is more specific than (F)).

- **(F)** Takes a single argument: a function
- **(F2)** Takes a single argument: a function that itself takes two arguments
- **(+)** Takes more than one argument

After each function, write (F), (F2), or (+):

- `map`
- `filter`
- `exists?`
- `all?`
- `curry`
- `uncurry`
- `foldl`
- `foldr`

You are now ready to tackle most parts of exercise 14.

4. Review the difference between `foldr` and `foldl` in section 2.8.1. You may also find it helpful to look at their implementations in section 2.8.3, which starts on page 130; the implementations are at the end.

   (a) Do you expect \((\text{foldl} + 0 \ '1\ 2\ 3\)) and \((\text{foldr} + 0 \ '1\ 2\ 3\)) to be the same or different?

   (b) Do you expect \((\text{foldl} \ \text{cons} \ '() \ '1\ 2\ 3\)) and \((\text{foldr} \ \text{cons} \ '() \ '1\ 2\ 3\)) to be the same or different?

   (c) Look at the initial basis, which is summarized on 156. Give one example of a function, other than + or cons, that can be passed as the first argument to `foldl` or `foldr`, such that `foldl` always returns exactly the same result as `foldr`.

   (d) Give one example of a function, other than + or cons, that can be passed as the first argument to `foldl` or `foldr`, such that `foldl` may return a different result from `foldr`.

You are now ready to tackle all parts of exercises 14 and 15.
5. Review function composition and currying, as described in section 2.7.2, which starts on page 125. Then judge the proposed algebraic laws below, which propose equality of functions, according to these rules:

- Assume that names curry, o, <, *, cons, even?, and odd? have the definitions you would expect, but that m may have any value.
- Each law proposes to equate two functions. If the functions are equal—which is to say, when both sides are applied to an argument, they always produce the same result—then mark the law Good. But if there is any argument on which the left-hand side produces different results from the right, mark the law Bad.

Mark these laws:

- \(((\text{curry } <) \ m) \ == \ (\lambda (n) \ (< m n))\)
- \(((\text{curry } <) \ m) \ == \ (\lambda (n) \ (< n m))\)
- \(((\text{curry cons}) \ 10) \ == \ (\lambda (xs) \ \text{cons} \ 10 \ xs)\)
- \((\text{o odd?} \ (\lambda (n) \ (* 3 n))) \ == \ \text{odd?}\)
- \((\text{o even?} \ (\lambda (n) \ (* 4 n))) \ == \ \text{even?}\)

You are now ready to tackle the first three parts of exercise 19, as well as problem M below.

Programming and Proof (90 percent)

Overview

For this assignment, you will do Exercises 14 (b-f,h,j), 15, and 19, from pages 209 to 212 of Build, Prove, and Compare, plus the exercises A, G1, G2, G3, M, and O below.

A summary of the initial basis can be found on page 156. A copy was handed out in class—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 internal function written with \(\lambda\) should be accompanied by a contract, but internal functions cannot be unit-tested. Algebraic laws are required only where noted below.

Book problems

14. Higher-order functions. Do exercise 14 on page 209 of Build, Prove, and Compare, parts (b) to (f), part (h), and part (j). You must not use recursion—solutions using recursion will receive No Credit. This restriction applies only to code you write. For example, gcd, which is defined in the initial basis, may use recursion.

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 sections 2.8.1 and 2.8.2 starting on page 128. For material on curry, see section 2.7.2, which starts on page 125.

15. Higher-order functions. Do exercise 15 on page 210. **You must not use recursion**—solutions using recursion will receive No Credit. As above, this restriction applies only to code 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\(^2\) 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 128. You may also find it helpful to study the implementations of `foldl` and `foldr` in section 2.8.3, which starts on page 130; the implementations are at the end. Information on lambda can be found in section 2.7, on pages 118 to 121.

19. Functions as values. Do exercise 19 on page 212 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 algebraic laws:

\[
\begin{align*}
\text{(member? } x \text{ (add-element } x \text{ s)} \text{) } &= \ldots \\
\text{(member? } x \text{ (add-element } y \text{ s)} \text{) } &= \ldots, \text{ where } \text{(not (equal? } y \text{ x)} \\
\text{(member? } x \text{ (union } s1 \text{ s2)) } &= \ldots \\
\text{(member? } x \text{ (inter } s1 \text{ s2)) } &= \ldots \\
\text{(member? } x \text{ (diff } s1 \text{ s2)) } &= \ldots
\end{align*}
\]

- 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 2.16.6 on page 191. In particular, be sure your code includes this record definition:

\[
\text{(record set-ops (empty member? add-element union inter diff))}
\]

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:

\[
\begin{align*}
\text{(define set-ops-from (eq?)} \\
\quad \text{(let ([empty } \ldots])}
\end{align*}
\]

\(^2\)In your README, please identify this credit as EXACT-EXISTS.
(make-set-ops empty member? add union inter diff))

Fill in each ... with your own implementations. Each implementation is like 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 (procedure? 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 nullset (set-ops-empty atom-set-ops))
(val member? (set-ops-member? atom-set-ops))
(val add-element (set-ops-add-element atom-set-ops))
(val union (set-ops-union atom-set-ops))
(val inter (set-ops-inter atom-set-ops))
(val diff (set-ops-diff atom-set-ops))

Related reading: For functions as values, see the examples of lambda in the first part of section 2.7 on page 118. For function composition and currying, see section 2.7.2. For polymorphism, see section 2.9, which starts on page 132.

Relating imperative code to functional code

A. Good functional style. The Impcore-with-locals function

(define f-imperative (y) (locals x)
  (begin
    (set x e)
    (while (p? x y)
      (set x (g x y)))
    (h x y)))

is in a typical imperative style, with assignment and looping. Write an equivalent µScheme function f-functional that doesn’t use the imperative features begin (sequencing), while (goto), and set (assignment).

• Assume that p?, g, and h are free variables which refer to externally defined functions.
• Assume that e is an arbitrary expression.
• Use as many helper functions as you like, as long as they are defined using let or letrec and not at top level.
• You need not write any algebraic laws.
Hint #1: If you have trouble getting started, rewrite while to use if and goto. Now, what is like a goto?

Hint #2: (set x e) binds the value of e to the name x. What other ways do you know of binding the value of an expression to a name?

Don’t be confused about the purpose of this exercise. The exercise is a thought experiment. We don’t want you to write and run code for some particular choice of g, h, p?, e, x, and y. Instead, we want you write a function that works the same as f-imperative given any choice of g, h, p?, e, x, and y. So for example, if f-imperative would loop forever on some inputs, your f-functional must also loop forever on exactly the same inputs.

Once you get your mind twisted in the right way, this exercise should be easy. The point of the exercise is not only to show that you can program without imperative features, but also to help you develop a technique for eliminating such features.

Related reading: No part of the book bears directly on this question. You’re better off reviewing your experience with recursive functions and perhaps the solutions for the Scheme assignment.

Graph problems

From COMP 15, you should be familiar with graphs and graph algorithms. In the next few problems you will work with an immutable representation of directed graphs: a graph is represented by an association list in which each node is associated with a list of its immediate successors. This representation is called a successors map. (It is a close cousin to the widely used “adjacency list.”)

For example, the ASCII-art graph

```
A --> B --> C
|    |
|    +-----------+
```

could be represented as a successors map by `(('([A (B C)] [B (C)] [C ()]).

Note: The graph problems below can be solved using only first-order functions. But you will find the problems much easier if you use let, lambda, and either of the fold functions.

Related reading: The previous assignment. The definitions of equal? in section 2.3.1 (basic recursive functions on lists). Material on association lists in section 2.3.6.

G1. List of edges. An edge is represented by a record

```
(make-edge N1 N2)
```

where N1 and N2 are nodes. Define make-edge using the following record definition:

```
(record edge [from to])
```

Function edge-list consumes a graph represented as a successors map and returns a list of all the edges in the graph. Edges may be listed in any order. For example, here are acceptable responses for a list of the edges in the graph pictured above:

```
(list3 (make-edge A B) (make-edge B C) (make-edge A C))
(list3 (make-edge A B) (make-edge A C) (make-edge B C))
```
Define function edge-list. Algebraic laws are optional, but unit tests are required.

Hints:

- You can solve this problem with algebraic laws, but you need to work at a high level of abstraction. Start by observing that a successors map has one of these two forms:
  - '()
  - (bind node successors graph), where node is a node, successors is a list of nodes, and graph is a graph represented as a successors map

  In the second form, extract node and successors using predefined functions alist-first-key and alist-first-attribute, and extract graph using cdr.

- There are plenty of lists in this problem. You will have an easier time if you find a way to use the predefined list functions, together with something you define that can add an edge to a list of edges.

- By 105 standards, the solution to this problem requires a lot of code. To keep it manageable, use let or let*.

G2. Graph-building: adding an edge. Function add-edge takes two arguments: an edge made with make-edge and a graph that is represented as a successors map. It returns a new graph that is like the original, except that the new graph has had the given edge added to it. Depending on whether the from node already appears in the graph, it may have to be added. (Determine its appearance using equal?)

For any edge e and graph g, function add-edge satisfies this algebraic law:

\[(\text{permutation?} \ (\text{cons} \ e \ (\text{edge-list} g))) \Rightarrow \ (\text{edge-list} \ (\text{add-edge} \ e \ g)))\]

Implement add-edge, and in addition, write the following:

- At least one unit test using check-assert and the law above
- At least one unit test using check-expect with an empty graph
- At least one unit test using check-expect with a nonempty graph

You may include the implementation of permutation? from the solutions to the previous homework.

Here are our requirements for algebraic laws:

- Each recursive function you define must be specified using algebraic laws.
- If none of your functions are recursive, you need not write any algebraic laws.

Hint: We know of at least two entirely different ways of coding add-edge:

- The first way is to treat the association list (that is, the successors map) entirely as an abstraction. That is, use only find, bind, and the “laws of association lists” shown in lecture—never look directly at the representation. To succeed in this way, you will have to understand what happens when you call find on a key that is not there.

- The second way is to get down in the weeds with the representation of the association list. You will wind up using car and cdr, and if you are smart you will also use alist-first-key and alist-first-attribute.
We recommend coding the first way, and we recommend avoiding recursion.

**G3. Graph update: removing a node.** Calling `(remove-node node graph)` returns a new graph that is like `graph`, but with all references to `node` removed:

- No value equal? to `node` appears as a key in the representation.
- No value equal? to `node` appears as the successor of any other node.

If the original graph does not mention `node`, then `(remove-node node graph)` returns a new graph that is equal? to the original.

Implement remove-node. For full credit, implement remove-node without using any recursion. If you do choose to use recursion, specify each recursion function by giving algebraic laws.

**Calculational reasoning about functions**

**M. Reasoning about higher-order functions.** Using the calculational techniques from Section 2.4.5, which starts on page 107, prove that

\[
(o \ ((\text{curry} \ \text{map}) \ f) \ ((\text{curry} \ \text{map}) \ g)) = ((\text{curry} \ \text{map}) \ (o \ f \ g))
\]

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

Take the following laws as given:

\[
\begin{align*}
(o \ f \ g) \ x &= (f \ (g \ x)) & ; \text{apply-compose law} \\
((\text{curry} \ f) \ x \ y) &= (f \ x \ y) & ; \text{apply-curried law}
\end{align*}
\]

Using these laws should keep your proof relatively simple.

**Related reading:** Section 2.4.5. 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.

**Ordered lists**

**O. Ordered lists.** I said in class that in most cases, a function that consumes lists uses the obvious inductive structure on lists: a list either empty or is made with `cons`. Here is a problem that requires a more refined inductive structure.

**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 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 is ordered by precedes?.
- A singleton list is ordered by precedes?.
- A list of 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?.

```scheme
-> ((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 structure of your function should be informed by the structure of the inductive definition of what it means for a list to be ordered by a relation. To elicit that structure, write algebraic laws.
- For the code itself, you will need `letrec`.
- 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.

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

**Related reading:** Section 2.9, which starts on page 132. Especially the polymorphic sort in section 2.9.2—the `lt?` parameter to that function is an example of a transitive relation. Section 2.7.2. Example uses of `map` in section 2.8.1. The definition of `map` in section 2.8.3.

**What and how to submit**

You must submit four 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
- A `cqs.hofs.txt` containing the reading-comprehension questions with your answers edited in
- A PDF files `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 14 (b-f,h,j), 15, 19, A, G1, G2, G3, and O. You must precede each solution by a comment that looks like something like this:

```scheme
;;
;; Problem A
;;
```

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:

```scheme
(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:

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

I’ve changed the name of the internal function, but the only other things that are different is that I have removed the formal parameter from the `lambda` and I have removed the second actual parameter from the call sites. I 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 functions length and so on in exercise 15. 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.

---

3../coding-rubric.html
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; don’t redefine them.

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

**Functional correctness**

In addition to the usual testing, we’ll evaluate the correctness of your translation in problem A. We’ll also want appropriate list operations to take constant time.
Correctness

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>• The translation in problem A is correct.</td>
<td>• The translation in problem A is almost correct, but an easily identifiable part is missing.</td>
<td>• The translation in problem A is obviously incorrect,</td>
</tr>
<tr>
<td>• Your code passes every one of our stringent tests.</td>
<td>• Testing reveals that your code demonstrates quality and significant learning, but some significant parts of the specification may have been overlooked or implemented incorrectly.</td>
<td>• Or course staff cannot understand the translation in problem A.</td>
</tr>
<tr>
<td>• Testing shows that your code is of high quality in all respects.</td>
<td>• Testing reveals your work to be substantially incomplete, or shows serious deficiencies in meeting the problem specifications (serious fault).</td>
<td>• Testing suggests evidence of effort, but the performance of your code under test falls short of what we believe is needed to foster success.</td>
</tr>
</tbody>
</table>

Performance

<table>
<thead>
<tr>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Empty lists are distinguished from non-empty lists in constant time.</td>
<td>• Distinguishing an empty list from a non-empty list might take longer than constant time.</td>
<td></td>
</tr>
</tbody>
</table>

Proofs and inference rules

For your calculational proof, use induction correctly and exploit the laws that are proved in the book.
<table>
<thead>
<tr>
<th>Proofs</th>
<th>Exemplary</th>
<th>Satisfactory</th>
<th>Must Improve</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>• Proofs that involve predefined functions appeal to their definitions or to laws that are proved in the book.</td>
<td>• Proofs involve predefined functions but do not appeal to their definitions or to laws that are proved in the book.</td>
<td>• A proof that involves an inductively defined structure, like a list or an S-expression, does <strong>not</strong> use structural induction, but structural induction is needed.</td>
</tr>
<tr>
<td></td>
<td>• Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction exactly where needed.</td>
<td>• Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction, even if it may not always be needed.</td>
<td></td>
</tr>
</tbody>
</table>