COMP 105 Assignment: Higher-Order Functions

Due Friday, February 22 at 5:59PM, **six hours before midnight**

Preliminaries: Setup, Interpreter, and Help With Tracing/Debugging

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 have updated `/comp/105/bin/uscheme` with the `&trace` facility described in part (b) of Exercise 50 on page 168 of Ramsey. You may find it useful.

Dire Warnings

The same warnings apply as last time: **none of the Scheme programs you submit may use any imperative features**. Banish `set`, `while`, `print`, 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 `print` while debugging, but they must not appear in any code you submit.)

As a substitute for assignment, use `let` or `let*`. *Also use `let` or `letrec` for ``helper'' functions.* **Except as noted below, do not define helper functions at top level**. When you do define inner helper functions, avoid passing as parameters values that are already available in the environment.

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

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

without any error messages. 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 readability).

Overview, organization, and what to submit

For this assignment, you will do Exercises **9 (b-g,i-j)**, **10**, **15**, and **22** from pages page 157–162 of Ramsey, plus the five exercises **A**, **M**, **P**, **S**, and **T** below. There are also extra-credit exercises of significant interest (and difficulty).

Place your solutions to Exercises **9 (b-g,i-j)**, **10**, **15**, **22** and exercises **A**, **P**, and **S**, as well as any extra credit that you choose to submit, in a file called `solution.scm`. (The solution to exercise **T** goes in its own file, `solver-tests.scm`.) Be sure to put the solutions in order and to precede each solution by a comment that looks like something like this:

```
;;
;; Problem P
;;
```

Place your solutions to Exercises **37** and **M** in a file called `semantics.pdf`. You can create this file using LaTeX or Lyx another mathematical word processor, or you can write your solution by hand and scan it.

- If you use LaTeX, the `mathpartir` package is invaluable for typesetting inference rules and derivations.
- If you scan your solutions, **please be sure all pages are oriented right-side up** in the resulting PDF. If pages are upside-down or sideways, we may ask you to resubmit.

On this assignment, **there is no pair programming**.

Details of all the exercises

A. *Good functional style*. The function

```
(define f-imperative (y) (x) ; x is a local variable
  (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 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.

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` should 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 to help you develop a technique for eliminating such features. You'll use this technique again later on.

My estimate of difficulty: most students find this exercise hard (though there is very little code)

9, 10. *Higher-order functions*. Do Exercise 9 on page 157 of Ramsey, parts(b) to (g), part (i), and part (j). Do Exercise 10 on page 158. 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 in the initial basis, or `insert`, which is given, may use recursion.) *For Exercise 9 only*, you may define helper functions at top level.

For Exercise 10 you get full credit if your implementations return correct results. You get **EXTRA CREDIT** (EXACT-EXISTS) if you can duplicate `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.

My estimate of difficulty: medium to easy (the first one or two are medium difficulty, but because the problems are so similar, once you have any one or two, the rest are easy)

M. *Reasoning about higher-order functions*. Using the calculational techniques from Section 3.4.5, 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.

Take the following laws as given:

```
((o f g) x) == (f (g x))        ; apply-compose law
(((curry f) x) y) == (f x y)    ; apply-curried law
```

Using these laws should keep your proof relatively simple.

My estimate of difficulty: medium

P. *Even more higher-order functions*. Write a function `palindrome?` that tells whether a list of symbols is a palindrome, that is, it spells the same words both forwards and backwards. Here's the catch: you have to ignore the dash used to separate words. Examples:

```
-> (palindrome? '(m a d a m - i m - a d a m))
#t
-> (palindrome? '(w a s - t h a t - a - r a t - i - s a w))
#f
-> (palindrome? '(w a s - i t - a - r a t - i - s a w))
#t
-> (palindrome? '(k u r r e m k a r m e r r u k))
#f
```

For **EXTRA CREDIT** (NOCASES), implement `palindrome?` without using an explicit `if`, either in `palindrome?` or in any helper function that *you* define.

My estimate of difficulty: easy

15. *Functions as values*. Do Exercise 15 on page 159 of Ramsey. When you code the third approach to polymorphism, please write a function `mk-set-ops`. This function should take one argument (the equality predicate) and should return a list of six values, in this order:

1. The empty set
2. Function `member?`
3. Function `add-element`
4. Function `union`
5. Function `inter`
6. Function `diff`

My estimate of difficulty: hard, because it requires a new way of thinking. Once you learn to think that way, the functions are easy.

S. *Higher-order, polymorphic sorting*. Using `filter` and `curry`, define a function `qsort` that, when passed a binary comparison function (like <), returns a Quicksort function. So, for example,

```
-> ((qsort <) '(6 9 1 7 4 14 8 10 3 5 11 15 2 13 12))
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)

-> ((qsort >) '(6 9 1 7 4 14 8 10 3 5 11 15 2 13 12))
(15 14 13 12 11 10 9 8 7 6 5 4 3 2 1)
```

You will also find it helpful to use the function-composition function `o`.

If you are not familiar with Quicksort, we have prepared a short Quicksort handout online.

Your Quicksort should not use the `append` function in any of its disguises. In other words, don't copy cons cells unnecessarily. (If you can't figure this part out, go ahead and use `append`; you will get partial credit.) *Hint #1: Use method of accumulating parameters covered in class when we discussed `revapp`. That is, think about writing a helper function that takes at least two arguments: a list `l` to be sorted and another list `tail` to be appended to the sorted list `l`.*

Hint #2: What part of Quicksort could `filter` and `o` help with?

Your code should use as few helper functions as possible. In particular, if you count up the number of occurrences of `define` and `lambda`, they should total at most three. (And if you give up and use `append`, that should save you a

`lambda`.) If you need more lambda abstractions, you are doing something wrong. As usual, any helper functions should be defined internally using `let` or `letrec`, not at top level.

Remember to **give a brief explanation of why your recursive sort routine terminates**. *If you write more than a dozen lines of code for this exercise, you're probably in trouble.*

> (For the bloody-minded among you, the C standard library specifies a higher-order Quicksort routine. How short an implementation can you write in C? How many more bugs did you find in your C version than in your Scheme version? How much longer did it take you? Do you find the answers surprising when you compare your experience with C to your experience with Scheme? *No credit is being offered for the answers to any of these C-related questions. I include them only so you can torture your friends who haven't had this course...* In case you wanted to know, P. J. Plauger has written a pretty good Quicksort in about 65 lines of ANSI standard C. He is quite careful about efficiency issues, like bounding use of the call stack.)

Here are some exacting test cases:

```
((qsort <) '(1 1 1))
((qsort <=) '(1 1 1))
((qsort <) '())
```

You might also try using qsort to sort a list of lists by putting the shortest lists first.

My Quicksort is 11 lines of μScheme.

My estimate of difficulty: medium

22. *Continuation-passing style*. Do Exercise 22 on page 162 of Ramsey. Don't overlook the possibility of deeply nested formulas with one kind of operator under another!
You must define a function `make-formula-true` which takes three parameters: a formula, a failure continuation, and a success continuation. The failure continuation should not accept any arguments, and the success continuation should accept two arguments: the first is the current (and perhaps partial) solution, and the second is a "resume" continuation.
My solution to this exercise is under 50 lines of μScheme.

My estimate of difficulty: hard

T. *Testing your solver*. In file `solver-tests.scm`, submit three test cases that together exercise *all* the capabilities of your solver. These test cases should be in their own file, and they should contain two `val` bindings for each test case: `f1` should be the formula input the the solver, and `s1` should be either a satifying assignment, or if no satisfying assignment exists, then it should be the symbol `no-solution`. If, for example, I wanted to code the test case that appears on page 112 of the book, I might write

```
(val f1 '(and (or x y z) (or (not x) (not y) (not z)) (or x y (not z))))
(val s1 '((x #t) (y #f)))
```

As another test case, I might write

```
(val f2 '(and x (not x)))
(val s2 'no-solution)
```

Be sure to consider combinations of the various Boolean operators. Explain *why* these particular test cases are important—your test cases must not be too complicated to be explained.

We hope to run every submitted solver on every test case. Your goal should be to design test cases that cause other solvers to fail.

Extra Credit

Extra credit (FIVES). *Programs as data*. To deepen your understanding of LISP and Scheme, here is a toy example of the kind of symbolic problem for which LISP is famous.

Consider the class of well-formed arithmetic computations using the numeral 5. These are expressions formed by taking the integer literal 5, the four arithmetic operators +, -, *, and /, and properly placed parentheses. Such expressions correspond to binary trees in which the internal nodes are operators and every leaf is a 5. Write a Scheme program to answer one or more of the following questions:

- What is the smallest positive integer than cannot be computed by an expression involving exactly five 5's?
- What is the largest prime number that can computed by an expression involving exactly five 5's?
- Exhibit an expression that evaluates to that prime number.

And, without implementing anything,

- Explain how you would change your implementation to use *exact* division instead of integer division.

Hints:

- You can build S-expressions that represent the arithmetic expressions in the exercise, and you can just call `eval` to find out what they evaluate to.
- This exercise involves an exhaustive search (for all numbers that can be computed with 5's), so good techniques are important. This is an excellent problem for dynamic programming (handout online).
- It will help you debug if you write a 'set of integer' implementation that keeps elements in order.
- You may want to speed up the search by writing a specialized version of eval.
- You may have to do something special to avoid division by zero. This is something of a pain in LISP, but you can cheat by specializing eval.
- Rational arithmetic is a good way to implement exact division.

Extra credit (FUNENV): In the theory, you've seen that we can represent environments as functions, not as assocation lists. You can do the same thing in code. If you used this new representation, how would you change the metacircular evaluator in Ramsey, Section 3.15? (You don't have to write the code, just explain how you would do it.) *Hint: you'll have to find a suitable value for the function to return in case the symbol isn't in the environment. Nil is probably not a good choice. In fact, nothing is a very good choice. This kind of dilemma motivates the use of exceptions in languages like CLU, ML, Modula-3, Ada, and C++.*

Extra credit (LAMBDA). `lambda` is more powerful than you might think. For extra credit, do any or all parts of Exercise 16 on page 159 of Ramsey. Because this problem redefines a bunch of standard constants, it needs to go in its own file, `lambda.scm`.

Test your work using the following scenario:

```
-> (define nth (n l)
      (if (= n 1) (car l)
                  (nth (- n 1) (cdr l))))
-> (val l (cons 'first (cons 'second (cons 'third nil))))
-> (nth 2 l)
second
-> (nth 3 l)
third
```

Hints:

- Perhaps you should use closures to represent cons cells and the empty list. Remember how we used `lambda` to store data when we did the the random-number generator in class.
- Given that `cons` should probably return a function (closure), try to make that function as simple as possible.

Avoid common mistakes

The most common mistakes on this assignment have to do with the Boolean-formula solver in Exercise 22. They are

- It's easy to handle fewer cases than are actually present in the exercise. You can avoid this mistake by considering all ways the operators `and`, `or`, and `not` can be combined pairwise to make formulas.
- It's easy to write near-duplicate code that handles essentially similar cases multiple times. This mistake is harder to avoid; I recommend that you look at your cases carefully, and if you see two pieces of code that look similar, try abstracting the similar parts into a function.
- It's easy to submit code with the wrong interface.

Another common mistake is passing 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)))
```

I've changed the name, but the only other things that are different is I've removed the formal parameter from the `lambda` and I've 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.

Another common mistake is to fail to redefine functions `length` and so on in Exercise 10. Yes, we really want you to provide new definitions that replace the existing functions, just as the exercise says.

Another common mistake is to put your answer to some part of 37 in your `solution.scm`. *All* parts of this answer, including Part B, go in `semantics.pdf`.

Another common mistake is to **forget to explain why `qsort` terminates**.

What to submit

Provide a `README` file, and in it, please do as follows:

- Please tell us with whom you collaborated
- Please tell us what problems you solved
- On each of the dimensions *documentation*, *naming*, *structure*, *form*, and *correctness* listed below, please let us know whether you believe your work was Exemplary, Satisfactory, or whether it needs improvement.
- Please tell us how many hours you spent on the assignment

If you want, include any insights you may have had about the exercises, but detailed remarks about your solutions are probably best left to comments in the source code.

If you wish, you may also turn in a file named `transcript.txt` that contains test cases for your solutions. You don't have to give us test cases; the test cases shown above are there to help you, not to make more work for you. If you do show test cases, please cut and paste a transcript of your interactions with the interpreters, just like the transcripts from the book, as in the following example:

```
-> (count    'a '(1 b a (c a)))
```

```
1
-> (countall 'a '(1 b a (c a)))
2
```

When you are ready, run `submit105-hofs` to submit your work, which should include the following files:

- `README`: This documentation file is mandatory.
- `solution.scm`: This source file is mandatory.
- `solver-tests.scm`: This source file is mandatory.
- `lambda.scm`: This source file is optional; if you've chosen to do the LAMBDA extra credit (Exercise 16 on page 159), put your answers here. (Your answers to Exercise 16 must *not* go in `solution.scm`.)
- `semantics.pdf`: This source file is mandatory; you may prepare it by computer, or you can scan a handwritten solution. **Please do not submit photos taken with cell-phone cameras**; they are blurry and hard to read. And as noted above, **please be sure all images in the PDF are right-side up.**
- `transcript.txt`: This optional file can contain your test cases.

How your work will be evaluated

Structure and organization criteria

Most of these you have seen before. As always, we emphasize **contracts** and **naming**. InÂ 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**.

There are a few new criteria around Quicksort and around the use of basis functions.

	Exemplary	Satisfactory	Must improve
Form	• Code is laid out in a way that makes good use of scarce vertical space. Blank lines are used judiciously to break large blocks of code into groups, each of which can be understood as a unit. • All code respects the offside rule • Indentation is consistent everywhere. • **New:** Indentation leaves most code in the left half or middle part of the line. • No code is commented out. • Solution file contains no distracting test cases or print statements.	• Code has a few too many blank lines. • Code needs a few more blank lines to break big blocks into smaller chunks that course staff can more easily understand. • The code contains one or two violations of the offside rule • In one or two places, code is not indented in the same way as structurally similar code elsewhere. • **New:** Indentation pushes significant amounts of code to the right margin. • Solution file may contain clearly marked test *functions*, but they are never executed. It's easy to read the code without having to look at the test functions.	• Code wastes scarce vertical space with too many blank lines, block or line comments, or syntactic markers carrying no information. • Code preserves vertical space too aggressively, using so few blank lines that a reader suffers from a "wall of text" effect. • Code preserves vertical space too aggressively by crowding multiple expressions onto a line using some kind of greedy algorithm, as opposed to a layout that communicates the syntactic structure of the code. • In some parts of code, every single line of code is separated form its neighbor by a blank line, throwing away half of the vertical space (**serious fault**). • The code contains three or more violations of the offside rule • The code is not indented consistently.

			• **New:** Indentation pushes significant amounts of code so far to the right margin that lots of extra line breaks are needed to stick within the 80-column limit. • Solution file contains code that has been commented out. • Solution file contains test cases that are run when loaded. • When loaded, solution file prints test results.
Naming	• Each function is named either with a noun describing the result it returns, or with a verb describing the action it does to its argument. (Or the function is a predicate and is named as suggested below.) • In a function definition, the name of each parameter is a noun saying what, in the world of ideas, the parameter represents. • Or the name of a parameter is the name of an entity in the problem statement, or a name from the underlying mathematics. • Or the name of a parameter is short and conventional. For example, a magnitude or count might be `n` or `m`. An index might be `i`, `j`, or `k`. A pointer might be `p`; a string might be `s`. A variable might be `x`; an expression might be `e`. A list might be `xs` or `ys`. • Names that are visible only in a very small scope are short and conventional.	• Functions' names contain appropriate nouns and verbs, but the names are more complex than needed to convey the function's meaning. • Functions' names contain some suitable nouns and verbs, but they don't convey enough information about what the function returns or does. • The name of a parameter is a noun phrase formed from multiple words. • Although the name of a parameter is not short and conventional, not an English noun, and not a name from the math or the problem, it is still recognizable---perhaps as an abbreviation or a compound of abbreviations. • Names that are visible only in a very small scope are reasonably short.	• Function's names include verbs that are too generic, like "calculate", "process", "get", "find", or "check" • Auxiliary functions are given names that don't state their contracts, but that instead indicate a vague relationship with another function. Often such names are formed by combining the name of the other function with a suffix such as `aux`, `helper`, `1`, or even `_`. • Course staff cannot identify the connection between a function's name and what it returns or what it does. • The name of a parameter is a compound phrase phrase which could be reduced to a single noun. • The name of some parameter is not recognizable---or at least, course staff cannot figure it out. • The name of a list parameter is neither a plural noun form nor a conventional name like `xs` or `ys`. • Long names are used in very small scopes (exception granted for some function parameters). • Very short names are used with global scope.
Documentation	• The contract of each function is clear from the function's name, the names of its parameters, and	• A function's contract omits some parameters.	• A function is not named after the thing it returns, and the function's documentation does not say what it

	perhaps a one-line comment describing the result. • When names are not enough, each function is documented with a contract that explains what the function returns, in terms of the parameters, which are mentioned by name. • From the name of a function, the names of its parameters, and the accompanying documentation, it is easy to determine how each parameter affects the result. • Documentation appears consistent with the code being described. • As an alternative to internal documentation, a function's documentation may refer the reader to the problem specification where the function's contract is given.	• A function's documentation mentions every parameter, but does not specify a contract. • A function's documentation includes information that is redundant with the code, e.g., "this function has two parameters." • A function's contract omits some constraints on parameters, e.g., forgetting to say that the contract requires nonnegative parameters.	returns. • A function's documentation includes a narrative description of what happens in the body of the function, instead of a contract that mentions only the parameters and result. • A function's documentation neither specifies a contract nor mentions every parameter. • There are multiple functions that are not part of the specification of the problem, and from looking just at the names of the functions and the names of their parameters, it's hard for us to figure out what the functions do. • Documentation appears inconsistent with the code being described.
Structure	• Short problems are solved using simple anonymous `lambda` expressions, not named helper functions. • **New:** Quicksort does not use `append` and is implemented using at most **three** `define` and `lambda`, in any combination. • **New:** Or, Quicksort uses `append` and is implemented using at most **two** `define` and `lambda`, in any combination. • **New:** Quicksort uses one `null?` test and one `if` • **New:** Quicksort has a very solid explanation for why it terminates. • **New:** Or, Quicksort has a believable explanation for why it terminates. • When possible, inner functions use the parameters and `let`-bound names of outer	• Most short problems are solved using anonymous lambdas, but there are some named helper functions. • **New:** Quicksort is implemented using more than three `define` and `lambda`, in any combination. • **New:** Or, Quicksort uses `append` and is implemented using three `define` and `lambda`s, in any combination. • **New:** Quicksort uses up to two `null?` tests and up to two `if`s. • **New:** Quicksort mentions termination. • 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. • Course staff have to work to tell whether the code is correct or incorrect.	• Most short problems are solved using named helper functions; there aren't enough anonymous `lambda` expressions. • **New:** Quicksort uses more than two `null?` tests or more than two `if`s. • **New:** Or, Quicksort does not use *any* `null?` tests or `if`s (**serious fault**). • **New:** Quicksort does not mention termination. • Helper functions are defined at top level. • From reading the code, course staff cannot tell whether it is correct or incorrect. • From reading the code, course staff cannot easily tell what it is doing. • There's about twice as much code

	functions directly. • Helper functions are defined internally using `let`, `let*`, or `letrec`. • The code of each function is so clear that, with the help of the function's contract, course staff can easily tell whether the code is correct or incorrect. • There's only as much code as is needed to do the job. • **New:** The initial basis of Î¼Scheme is used effectively.	• There's somewhat more code than is needed to do the job. • **New:** Functions in the initial basis, when used, are used correctly.	as is needed to do the job. • **New:** Functions in the initial basis are redefined in the submission.
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.

Cost and correctness of your code

We'll be paying some attention to cost as well as correctness.

	Exemplary	Satisfactory	Must improve
Correctness	• The translation in problem A is correct. • Your code passes every one of our stringent tests. • Testing shows that your code is of high quality in all respects. • **New:** File `solver-tests.scm` contains exactly 6 `val` bindings and no other code. • **New:** In file `solver-tests.scm`, values `s1`, `s2`, and `s3` are either satisfying assignents or the symbol `no-solution`. • **New:** In file `solver-tests.scm`, values `f1`, `f2`, and `f3` represent valid formulas.	• The translation in problem A is almost correct, but an easily identifiable part is missing. • Testing reveals that your code demonstrates quality and significant learning, but some significant parts of the specification may have been overlooked or implemented incorrectly.	• The translation in problem A is obviously incorrect, • Or course staff cannot understand the translation in problem A. • 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**). • **New:** File `solver-tests.scm` contains other code besides the 6 `val` bindings requested. • **New:** In file `solver-tests.scm`, value `s1`, `s2`, or `s3` claims to be a satisfying assignment, but it isn't.

			• **New:** Or, in file `solver-tests.scm`, value `s1`, `s2`, or `s3` claims there is no solution, but the corresponding formula *does* have a solution. • **New:** In file `solver-tests.scm`, values `f1`, `f2`, or `f3` does not represent a valid formula.

Proofs and inference rules

These are the same criteria as before, with a little extra emphasis on using structural induction correctly.

	Exemplary	Satisfactory	Must improve
Proofs	• **New:** Proofs that involve predefined functions appeal to their definitions or to laws that are proved in the book. • **New:** Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction exactly where needed.	• **New:** Proofs involve predefined functions but do not appeal to their definitions or to laws that are proved in the book. • **New:** Proofs that involve inductively defined structures, including lists and S-expressions, use structural induction, even if it may not always be needed.	• **New:** 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.