# CSU 290 (Spring 2008)

This is the resource page for Section 2 of Logic and Computation, CSU 290 (Spring 2008). Welcome.

The purpose of this page is to simply complete the lectures, partly to keep us in sync with the other section, partly because there is only so much time and much interesting things to discuss.

These notes are meant to supplement the lectures, the readings and the lecture notes (available on the course web site) and not replace them.

Feel free to comment at the bottom of the page if you have any questions or anything is unclear. (Select "Add Comment" under "Add Content" down below.) I will generally merge your comments and questions into the main text when I get to them.

**What you should do right now**: Make sure you are logged into the wiki. (Select "Log In" under "Your Account" below if not, use you CCIS credentials to log in.) Look at the bottom of the page, under "Other Features", top right icon is an envelope. Press that button. That will register you as *watching this page*. Basically, every time the page changes, you get an email saying so. Keeps you from having to check the page regularly.

---

## February 15: Formal Proof for `~(t=nil)`

Here is a complete formal proof for `~(t=nil)` using the axioms I gave in class. (This is the proof that I started on Monday, but ran out of time 5 steps into the proof.)

You should be able to understand all the steps in that proof, if not be able to come up with the proof yourself. We will not ask you to come up with these kind of formal proofs in this course. Instead, we will use the more informal "chain of equals" or "chain of relations" approach I presented in class.

In the proof below, I write ~A for "not A" and `A & B` for "A and B".

```
1. ~(equal (car (cons x y) x) x) = nil        [ axiom for equal/car/cons
2. ~(equal (car (cons a b) a) a) = nil)       [ instantiation of 1 with x->a, y->b
```

Okay, so 2 is something we will use later on. Let's remember it.

```
3. ~(x = y) => (equal x y) = nil              [ axiom for equal
4. ~(car (cons a b)) = a =>
         (equal (car (cons a b)) a) = nil     [ instantiation of 3 with x->(car (cons a b)), y->a
5. ~(~(car (cons a b)) = a)                   [ by MT applied to 4 and 2
6. ~(~A) => A                                 [ tautology of Boolean logic
7. ~(~(car (cons a b)) = a) =>
      (car (cons a b)) = a                    [ instantiation of 6 with A->(car (cons a b))
8. (car (cons a b)) = a                       [ by MP applied to 7 and 5
9. x=y => (equal x y) = y                     [ axiom for equal
10. (car (cons a b)) = a =>
        (equal (car (cons a b)) a) = t        [ instantiation of 9 with x->(car (cons a b)),y->a
11. (equal (car (cons a b)) a) = t            [ by MP applied to 10 and 8
```

Excellent. So now we can see that 2 and 11 together essentially give us that `t /= nil`.
We just have to work some to get that to come out of the proof.

```
12. (A & B => C) => (A & ~C => ~B)            [ tautology of Boolean logic (check it!)
13. ((equal (car (cons a b)) a) = t
      & t = nil =>
          (equal (car (cons a b)) a) = nil))  [ instantiation of 12 with
     => ((equal (car (cons a b)) a) = t       [  A->(equal (car (cons a b)) a) = t
        & ~(equal (car (cons a b)) a) = nil)) [  B->(t=nil)
            => ~(t = nil))                    [  C->(equal (car (cons a b)) a) = nil
14. x = y & y = z => x = z                    [ axiom of transitivity
15. (equal (car (cons a b)) a) = t            [ instantiation of 14 with
     & t = nil =>                             [   x->(equal (car (cons a b)) a)
         (equal (car (cons a b)) a) = nil     [   y->t  and z->nil
16. (equal (car (cons a b)) a) = t            [ by MP applied to 13 and 15
    & ~(equal (car (cons a b)) a) = nil
          => ~(t = nil)
```

And we are just about done. We have both antecedents of the implication in 16.

```
17. ~(t=nil)                                  [ by MP applied to 16, 11, and 2
```

# February 20: Review of Proving Theorems

Here is what we have seen until now as far as proving theorems are concerned.

We reviewed Boolean logic and the concept of a tautology of Boolean logic. Recall that a tautology is just a Boolean formula that is always true, irrespectively of the truth values you give to the Boolean variables in the formula.

We shall take Boolean (aka propositional) reasoning for granted.

We introduced the ACL2 logic, as essentially Boolean logic augmented with the ability to reason about basic formulas of the form EXP1 = EXP2, where EXP1 and EXP2 are ACL2 expressions.

The ACL2 logic is defined formally using the following axioms and rules of inferences:

- Every tautology of Boolean logic is an axiom
- `x = x` (reflexivity)
- `x = y => y = x` (symmetry)
- `x = y /\ y = z => x = z` (transitivity)
- `x1 = y1 /\ ... /\ xn = yn => (f x1 ... xn) = (f y1 ... yn)` for every function symbol `f` of arity n
- From `F` derive `F/s` (where `F` is an arbitrary formula, and `s` is a substitution of ACL2 expressions for free variables in `F`)
- From `F` and `F => G` derive `G` (modus ponens)

We also have axioms corresponding to the different primitives of the language. The basic ones are:

- `x = y => (equal x y) = t`
- `x /= y => (equal x y) = nil` (where `/=` represents "not equal to")
- `x = nil => (if x y z) = z`
- `x /= nil => (if x y z) = x`
- `(car (cons x y)) = x`
- `(cdr (cons x y)) = y`
- `consp (cons x y) = t`
- `consp (nil) = nil`
- `(integerp x) = t => (consp x)=nil`

Additionally, we can take as axioms the basic properties of arithmetic that you have seen in high school and discrete maths. Thus, when proving theorems, you will be free to perform standard simplification of arithmetic.

A formal proof of `F` is a sequence of formulas such that every formula in the sequence is either an axiom or derived from previous formulas using an inference rule, and such that the last formula of the sequence is `F` itself.

I gave you a formal proof of `t /= nil` above. We shall not write a lot of formal proof, and you certainly will not have to write any of them.

Instead, we shall use some proof techniques to prove formulas that have a certain form. These techniques do not work for all formulas, but they work for enough of them to be interesting.

**If the formula you want to prove is of the form EXP1 = EXP2**, then one technique is to use a sequence of equalities to equate EXP1 and EXP2:

```
   EXP1
=
   ....
=
   EXP2
```

where all the equalities are justified using the axioms or the inference rules above. By transitivity of =, this clearly shows that EXP1 = EXP2.

Alternatively, it is sometimes easier to simplify both EXP1 and EXP2 into a common expression EXP3.

```
   EXP1
=
   ...
=
   EXP3


   EXP2
=
   ...
=
   EXP3
```

Again, it should be pretty clear that this proves that EXP1 = EXP2. (By transitivity of = once again.)

**If the formula you want to prove is of the form HYPS => EXP1 = EXP2**, where HYPS is some hypotheses (formulas), then you can use a variant of the above, that is, prove that EXP1 = EXP2 using a sequence of equalities (or simplify EXP1 and EXP2 to the same expression EXP3), except that when you derive the equalities you can in addition to the axioms and inference rules use the hypotheses in HYPS to help you.

---

## February 25: Solutions to Exam 3 Proof Questions

Recall the definitions:

```
(defun add99 (l)
  (if (endp l)
      nil
    (cons (cons 99 (car l)) (add99 (cdr l)))))

(defun same-lenp (l n)
  (if (endp l)
      t
    (if (equal (len (car l)) n)
        (same-lenp (cdr l) n)
        nil)))

(defun len (l)
  (if (endp l)
      0
    (+ 1 (len (cdr l)))))
```

Question 2. Prove `(consp x) => (len (car (add99 x))) = (+ (len (car x)) 1)`

```
  (len (car (add99 x)))
=  {def of add99 }
  (len (car (if (endp x) nil (cons (cons 99 (car x)) (add99 (cdr x))))))
=   { (endp x) = nil }
  (len (car (cons (cons 99 (car x)) (add99 (cdr x)))))
=   {car-cons axiom}
  (len (cons 99 (car x)))
=   {def of len }
  (if (endp (cons 99  (car x)))
      0
      (+ 1 (len (cdr (cons 99 (car x))))))
=   {endp(cons ...)=nil}
  (+ 1 (len (cdr (cons 99 (car x)))))
=  {cdr-cons axiom}
  (+ 1 (len (car x)))
```

Question 3. Prove `(endp x) => (same-lenp (add99 x) y)`

```
  (same-lenp (add99 x) (+ y 1))
=  {def of add99}
  (same-lenp (if (endp x) nil ...))
=   { (endp x) = t }
  (same-lenp nil (+ y 1))
=   {def of same-lenp }
  (if (endp nil) t ...)
=   { (endp nil)=t }
  t
```

Question 4. Prove `((same-lenp x y) /\ (consp x) /\ (same-lenp (add99 (cdr x)) (+ y 1))) => (same-lenp (add99 x) (+ y 1))`

(You can use the fact that `((same-lenp x y) /\ (consp x)) => (+ 1 (len (car x))) = (+ 1 y)`.)

```
  (same-lenp (add99 x) (+ y 1))
=  {def of add99}
  (same-lenp (if (endp x) nil (cons (cons 99 (car x)) (add99 (cdr x)))) (+ y 1))
=   {(endp x) = nil}
  (same-lenp (cons (cons 99 (car x)) (add99 (cdr x))) (+ y 1))
=    {def of same-lenp}
  (if (endp (cons (cons 99 (car x)) (add99 (cdr x))))
      t
    (if (= (len (car (cons (cons 99 (car x)) (add99 (cdr x))))) (+ y 1))
        (same-lenp (cdr (cons (cons 99 (car x)) (add99 (cdr x)))) (+ y 1))
      nil))
=  { (endp (cons ...)) = nil }
  (if (= (len (car (cons (cons 99 (car x)) (add99 (cdr x))))) (+ y 1))
      (same-lenp (cdr (cons (cons 99 (car x)) (add99 (cdr x)))) (+ y 1))
    nil)
=  { car-cons and cdr-cons axioms}
  (if (= (len (cons 99 (car x))) (+ y 1))
      (same-lenp (add99 (cdr x)) (+ y 1))
    nil)
=   { def of len }
  (if (= (if (endp (cons 99 (car x)))
             0
           (+ 1 (len (cdr (cons 99 (car x))))))  (+ y 1))
      (same-lenp (add99 (cdr x)) (+ y 1)))
      nil)
= { (endp (cons ...)) = nil
  (if (= (+ 1 (len (cdr (cons 99 (car x))))) (+ y 1)))
       (same-lenp (add99 (cdr x)) (+ y 1)))
       nil)
= { car-cons axiom }
  (if (= (+ 1 (len (car x)) (+ 1 y)))
       (same-lenp (add99 (cdr x)) (+ y 1)))
       nil)
=  { by fact above }
  (same-lenp (add99 (cdr x)) (+ y 1))
=  { by hypothesis }
  t
```

---

## February 26: On Termination

Here is the argument that, at least in Scheme, it is impossible to write a function that correctly determines whether another function terminates. We argue by contradiction.

Suppose that we could write a function (terminates? f x) that returns #t (true) when (f x) terminates, and returns #f (false) when (f x) does not terminate. I don't care how terminates? is written - just suppose you managed to write it. I will not show that you get something completely absurd as a result.

In particular, I can now write the following perfectly legal scheme function:

```
  (define (loop-forever) (loop-forever))

  (define (oops f x)
    (if (terminates? oops empty)
        (loop-forever)
        1))
```

My question: does (oops empty) terminate?

Let's see. It either does or doesn't. So let's consider both cases.

1. If (oops empty) terminates, then by assumption (terminates? oops empty) must be true, and therefore, by the code of oops, (oops empty) must loop forever, i.e., not terminate, contradicting that (oops empty) terminates.

1. if (oops empty) does not terminate, then by assumption (terminates? oops empty) must be false, and therefore by the code of oops, (oops empty) returns 1 immediately, i.e., terminates, contradicting that (oops empty) does not terminate.

Because we get a contradiction no matter what, it must be that what we initially assumed was wrong, i.e., that terminates? works correctly. In other words, we cannot write a correct terminates? function in Scheme.

It turns out that this argument can be made to work for any programming language, but you'll have to wait for CSU 390 to see it.

One approach to arguing that a recursive function terminates is to argue that we are making progress towards the base case. The design recipe that we gave you and that you learned in 211 is in fact meant to ensure exactly this, at least for the kind of functions that we have seen until now.

It is sometimes a little subtle to argue that you are indeed making progress towards termination. In particular, consider the following ACL2 function:

```
(defun foo (n)
   (cond ((zp n) 0)
         ((= n 1) 1)
         ((evenp n) (foo (/ n 2)))
         ((oddp n) (foo (+ n 1)))))
```

(Assume that we have correct definitions for `evenp` and `oddp`.) I claim that this terminates for all inputs. Clearly, if the input is not a natural number, it terminates immediately. (Why?) Otherwise, can we argue that we are making progress towards the base cases (either 0 or 1)? Here are two arguments: the first, the one raised in class on Monday, is that that even though when the input is odd the recursive call is done on a bigger number (that therefore does not immediately progresses towards the base case), (foo (+ n 1)) will be called on an even number, which means that at the following iteration, foo will be invoked on (/ (+ n 1) 2), which is smaller than n. Thus, even though at every step we are not making progress towards the base case, we are making progress towards the base case at either every step or every two steps, depending on whether the input is even or odd. That's actually sufficient to establish termination. (Why?)

Here's another way of thinking about it that actually shows progress towards termination at every step. Write the input as a binary number, and consider the following number <n> derived from the input n:

```
<n> = the number of 1's in the binary representation of n
      + 2 * (the length of n in binary)
      - the number of 0's in the binary representation of n to the right of the rightmost 1.
```

Thus,

- 4 is 100 in binary, so <4> = 1 + (2 * 3) - 2 = 1 + 6 - 2 = 5,
- 5 is 101 in binary, so <5> = 2 + (2 * 3) - 0 = 2 + 6 = 8,
- 6 is 110 in binary, so <6> = 2 + (2 * 3) - 1 = 2 + 6 - 1 = 7.

You can check that at every recursive step of the function, foo is called on an input n whose <n> is strictly smaller than the original input to the function. And we are making progress towards the base case, with <1> = 3.

However, this doesn't work for every function, even those that look similar to the above. One of the most famous examples is the following so-called Collatz function:

```
(defun collatz (n)
  (cond ((zp n) 0)
        ((= n 1) 1)
        ((evenp n) (collatz (/ n 2)))
        ((oddp n) (collatz (+ (* 3 n) 1)))))
```

It is unknown whether this function terminates on all inputs. Plot out a few calls for small values of n, and you'll get a sense for how long some of the iterations take. Termination has been checked for all natural numbers up to 10^16, but for all we know it fails to terminate on some input greater than 10^17. We just don't know enough about number theory to say anything else.

I posted some more mathematical references to this function here.

---

## February 27: The Final Bit of the Proof that `(len (add1 a)) = (len a)`

I went over it quickly at the end of lecture today, so it pays to look at it more slowly.

Recall the definitions I gave in class:

```
(defun len (l)
  (if (endp l)
      0
      (+ 1 (len (cdr l)))))

(defun add1 (l)
  (if (endp l)
      l
      (cons (+ 1 (car l))
            (add1 (cdr l)))))
```

I claimed that in order to prove that `(len (add1 a)) = (len a)`, it suffices to prove the following two theorems:

```
Add1EndP:    (endp a) => (len (add1 a)) = (len a)

Add1ConsP:   (consp a) /\ (len (add1 (cdr a))) = (len (cdr a)) => (len (add1 a)) = (len a)
```

We saw that these are actually quite easy to prove. (Do it again if you're shaky on the details.)

We need to argue, though, that `Add1EndP` and `Add1ConsP` together give you that `(len (add1 a)) = (len a)` for all possible `a` in the ACL2 universe.

We are going to argue this by contradiction. So assume that it is *not* the case that `(len (add1 a)) = (len a)` for all `a`.

Let A be the set of all values in the ACL2 universe for which `((len (add1 a))` is not equal to `(len a)`. By assumption, A is not empty.

Let `s` be an element of A that has smallest length amongst all elements of A. (If more than one element has smallest length, pick any of them.)

First off, we know that `s` cannot be an atom. Why? Because we proved `Add1EndP`, which says that atoms cannot be in A. So `s` must be a cons pair, that is, `(consp s) = t`.

Because `s` is a cons pair, `(cdr s)` must have length strictly less than that of `s`. Because `s` was chosen to have smallest length in A, then `(cdr s)` cannot be in A.

But because `(cdr s)` is not in A, then it must be that `(len (add1 (cdr s))) = (len (cdr s))`. (Recall how we defined A in the first place.)

So we know that `(consp s) = t`, and that `(len (add1 (cdr s))) = (len (cdr s))`. But we proved `Add1Consp`, which says that it must be the case that `(len (add1 s)) = (len s)`. And this contradicts the fact that `s` is in A in the first place!

We derive a contradiction, so what we assumed in the first place must be false - so there cannot be *any* `a` such that `(len (add1 a))` is not equal to `(len a)`, that is, `(len (add1 a)) = (len a)` for all `a` in the universe.

---

## March 11: Some Easy Proof Exercises

Consider the following definitions:

```
(defun andp (x y)
  (if x
      (if y t nil)
    nil))

(defun orp (x y)
  (if x
      t
      (if y t nil)))
```

Make sure you understand them. You can assume the following properties about `andp` and `orp`:

```
  (booleanp (andp x y))
  (andp x y) = (andp y x)
  (booleanp (orp x y))
  (orp x y) = (orp y x)
```

**EDIT**: Let me add the following properties that you can assume too:

```
(booleanp z) => (andp z t) = z
(booleanp z) => (andp t z) = z
(booleanp z) => (orp z nil) = z
(booleanp z) => (orp nil z) = z
```

(You can give a shot at proving these, but the proof is a bit different than what we're used to - it requires case analysis, that you can do using only Boolean reasoning. For now, just take the properties above as true. I will talk about the proof technique for this in some other entry below.)

Now, let's write a function that applies `andp` to a list of values, so that it returns true exactly when not all values in the list are nil.

```
(defun and-foldp (l)
  (if (endp l)
      t
    (andp (car l) (and-foldp (cdr l)))))
```

Note what happens when we evaluate `(and-foldp '())`, or when we evaluate `(and-foldp 5)`, or when we evaluate `(and-foldp '(t))`.

Let's do something similar for `orp`:

```
(defun or-foldp (l)
  (if (endp l)
      nil
    (orp (car l) (or-foldp (cdr l)))))
```

Again, note what happens when we evaluate `(or-foldp '())`, or when we evaluate `(or-foldp 5)`, or when we evaluate `(or-foldp '(t))`.

Let's prove some theorems. (Many of these are proof obligations obtained from the induction principle - can you identify the formula that these proof obligations end up proving?) Make sure that you understand what the theorem you are trying to prove is actually saying! Otherwise, this exercise is just meaningless symbol pushing, which is going to be excruciatingly boring.

- `(endp x) => (booleanp (and-foldp x))`
- `(consp x) & (booleanp (and-foldp (cdr x))) => (booleanp (and-foldp x))`
- `(endp x) => (booleanp (or-foldp x))`
- `(consp x) & (booleanp (or-foldp (cdr x))) => (booleanp (or-foldp x))`
- `(and-foldp (cons a b)) = (andp a (and-foldp b))`
- `(or-foldp (cons a b)) = (orp a (or-foldp b))`

Prove the following theorems, using the standard definition of `app`:

```
(defun app (x y)
  (if (endp x)
      y
    (cons (car x) (app (cdr x) y))))
```

- `(endp x) => (and-foldp (app x y)) = (andp (and-foldp x) (and-foldp y))`
- `(consp x) & (and-foldp (app (cdr x) y)) = (andp (and-foldp (cdr x)) (and-foldp y)) => (and-foldp (app x y)) = (andp (and-foldp x) (and-foldp y))`
- `(endp x) => (or-foldp (app x y)) = (orp (or-foldp x) (or-foldp y))`
- `(consp x) & (or-foldp (app (cdr x) y)) = (orp (or-foldp (cdr x)) (or-foldp y)) => (or-foldp (app x y)) = (orp (or-foldp x) (or-foldp y))`

**EDIT**: note that proving the above gives you that the following two theorems are true:

- `(and-foldp (app x y)) = (andp (and-foldp x) (and-foldp y))`
- `(or-foldp (app x y)) = (orp (of-foldp x) (or-foldp y))`

These may come in handy in the proofs of the next theorems.

For the next few theorems, you will want to prove the following lemma first:

- `(consp x) => (app (list (car x)) (cdr x)) = x`

(Recall that `(list z)` is just an abbreviation for `(cons z nil)`. You can use the following axiom for cons: `(consp x) => (cons (car x) (cdr x)) = x`.)

Use the standard definition of `rev`:

```
(defun rev (x)
  (if (endp x)
      nil
    (app (rev (cdr x)) (list (car x)))))
```

Prove the following theorems:

- `(endp x) => (and-foldp (rev x)) = (and-foldp x)`
- `(consp x) & (and-foldp (rev (cdr x))) = (and-foldp (cdr x)) => (and-foldp (rev x)) = (and-foldp x)`
- `(endp x) => (or-foldp (rev x)) = (or-foldp x))`
- `(consp x) & (or-foldp (rev (cdr x))) = (or-foldp (cdr x)) => (or-foldp (rev x)) = (or-foldp x)`

---

## March 17: Proof of `(true-listp x) => (rev (rev x)) = x`

Here is the proof of `(true-listp x) => (rev (rev x)) = x` that I did in class, complete with all auxiliary lemmas.

We proceed by induction on x, meaning that we have to prove the following proof obligations:

- `(endp x) & (true-listp x) => (rev (rev x)) = x`
- `(consp x) & ((true-listp (cdr x)) => (rev (rev (cdr x))) = (cdr x)) & (true-listp x) => (rev (rev x)) = x`

**Proof** of `(endp x) & (true-listp x) => (rev (rev x)) = x`:

```
  (rev (rev x))
=  { by def of rev, if axiom, hypothesis}
  (rev nil)
=  { by def of rev }
  nil
=  { by lemma 1 below, (endp x) & (true-listp x) => x = nil }
  x
```

This proof relies on the following lemma, which says that if we have a true list and it is an atom, then it must be `nil`. Note that we use this lemma to add a new result to the *context*, that is, to the set of known facts that include the hypotheses.

**Proof** of Lemma 1: `(endp x) & (true-listp x) => x = nil`

```
We first prove that (endp x) & (true-listp x) => (equal x nil) = t:

  t
=  {by hypothesis}
  (true-listp x)
=  {by def of true-listp, hypothesis, if axiom}
  (equal x nil)

This gives us that (equal x nil) = t, that is, x = nil (by the equal axiom).
```

Back to the main proof, where we now have to prove the inductive case:

**Proof** of `(consp x) & ((true-listp (cdr x)) => (rev (rev (cdr x))) = (cdr x)) & (true-listp x) => (rev (rev x)) = x`:

```
  (rev (rev x))
=  {by def of rev, hypothesis, if axiom}
  (rev (app (rev (cdr x)) (list (car x))))
=  {by Lemma 2 below}
  (app (rev (list (car x))) (rev (rev (cdr x))))
=  {by induction hypothesis, knowing that (true-listp (cdr x)) using Lemma 3 below}
  (app (rev (list (car x))) (cdr x))
=  {by def of rev}
  (app (list (car x)) (cdr x))
=  {by def of app}
  (cons (car x) (cdr x))
=  {by cons axiom, knowing that (consp x)}
  x
```

The proof uses two lemmas. Let's take care of the third lemma first.

**Proof** of Lemma 3: `(true-listp x) & (consp x) => (true-listp (cdr x))`:

```
  t
=  {by hypothesis}
  (true-listp x)
=  {by def of true-listp, hypothesis, if axiom}
  (true-listp (cdr x))
```

That was easy. Slightly harder is lemma 2, the one that Eric proved on the board today.

**Proof** of Lemma 2: `(rev (app a b)) = (app (rev b) (rev a))`:

```
By induction on a. (Why?)

Base case: (endp a) => (rev (app a b)) = (app (rev b) (rev a))

Let's prove that both sides equal a common value.

  (rev (app a b))
=  {by def of app, hypothesis, if axiom}
  (rev b)

  (app (rev b) (rev a))
=  {by def of rev, hypothesis, if axiom}
  (app (rev b) nil)
=  {by lemma (true-listp x) => (app x nil) = x, knowing that (rev b) is a true-listp by Lemma 4 below}
  (rev b)


Inductive case: (consp a) & (rev (app (cdr a) b)) = (app (rev b) (rev (cdr a))) =>
                     (rev (app a b)) = (app (rev b) (rev a))

Again, we prove that both sides equal a common value.

  (rev (app a b))
=  {by def of app, hypothesis, if axiom}
  (rev (cons (car a) (app (cdr a) b)))
=  {by def of rev, endp axiom, if axiom, car-cons axiom}
  (app (rev (app (cdr a) b)) (list (car a)))
=  {by induction hypothesis}
  (app (app (rev b) (rev (cdr a))) (list (car a))

  (app (rev b) (rev a))
=  {by def of rev, hypothesis, if axiom}
  (app (rev b) (app (rev (cdr a)) (list (car a))))
=  {by lemma (app a (app b c)) = (app (app a b) c)}
  (app (app (rev b) (rev (cdr a))) (list (car a)))
```

All that's missing now is Lemma 4, that says that `(rev b)` is a true list for any `b`. I left it as an exercise for you to prove for next time.

---

## April 9: Topics for Exam #6

Topics covered include everything we have learned about the ACL2s theorem prover, lectures 25 to 29 on the web page; of course, you should not forget the basics of proving theorems.

Here are the main things we will test.

1. Understanding ACL2s proof attempts
   That was the largest component of these last two weeks.
   In particular, understand the waterfall model (cf lecture 25), and be able to extract lemmas to be proved when ACL2s is stuck on a proof.
2. Understand rewriting
   Cf lecture 26 and the textbook, and see below.
3. Understand decision procedures for propositional reasoning and congruence closure
   Cf lecture 29 and see below.

## Rewriting

Recall that when you prove a theorem (possibly with hypotheses) with a conclusion of the form LHS = RHS, this defines a rewrite rule that rewrites occurrences of LHS into RHS. (Note the direction!) Variables in the LHS are taken to be placeholders, that can be substituted for when rewriting happens.

The sequencing of rewrites when rewriting happens is important. The two main rules are:

1. Rewriting occurs for inner expressions before outer expressions. Thus, if you have an expression `(f (g x))`, the rewriter will attempt to rewrite `(g x)` first, and then attempt to rewrite `(f ...)`.
2. Rewriting occurs with the **latest** rewrite rules defined tried first, all to way to the oldest rewrite rules. In other words, if two (or more) rewrite rules are applicable to a given expression, the most recently admitted rewrite rule is used.

## Decision Procedure for Propositional Reasoning

How does ACL2s decided whether a Boolean expression is a tautology or not? It could build a truth table, but it doesn't. (Well, actually, it almost does - it just uses a more efficient representation than a truth table, that contains essentially the same information.) Here's how it does it. For simplicity, here, I will only consider the case where there are no occurrence of functions others than `implies`, `and`, `not`, and `or` in the Boolean expression. (It's an interesting exercise to figure out what happens when you do allow other functions.)

First, it rewrites the Boolean expression so that it does not contain any `implies`, `and`, `or`, and `not`. It does so by replacing every occurrence of those functions by a corresponding `if` expression. Here are the replacements:

```
(implies X Y) -> (if X Y t)
(and X Y) -> (if X Y nil)
(or X Y) -> (if X t Y)
(not X) -> (if X nil t)
```

(Convince yourself that these are correct by trying out a few examples, that is, seeing whether `(or t nil)` evaluates to `t` when replaced by the corresponding `if` expression, etc.)
Thus, for example:

```
(and (not a) a) = (if (if a nil t) a nil)     # expanding out and and not
```

Once that is done, you are left with a bunch of `if` expressions. For reasons that will become clear, we want to simplify the *condition* of those `if` expressions as much as possible. In particular, we want all `if` to have simple variables as conditions. To do so, we will apply the following transformation to an `if` expression that has a non-variable as a condition. (If it's not a variable, it has to be an `if` expression itself.)

```
(if (if X Y Z) V W) -> (if X (if Y V W) (if Z V W))
```

(Convince yourself that you understand why applying this transformation does not change the value returned by the whole expression.)
Do this repeatedly until all `if` expressions have a variable (or `t` or `nil`) as a condition.
Continuing our example above:

```
(and (not a) b) = (if (if a nil t) a nil)    # expanding out and and not
                = (if a                      # applying transformation above
                      (if nil a nil)
                    (if t a nil))
```

Now, let's simplify things further.
First, take every `if` expression, from the *outermost* to the *innermost*:

1. if the expression is of the form `(if t X Y)`, replace it by `X`, and repeat going into `X`.
2. if the expression is of the form `(if nil X Y)`, replace it by `Y`, and repeat going into `Y`.
3. if the expression is of the form `(if v X Y)`, where `v` is a variable, replace every occurrence of `v` in `X` by `t` and every occurrence of `v` in `Y` by `nil`, and repeat (going into `X` and `Y`).

Continuing our example:

```
(and (not a) b) = (if (if a nil t) a nil)    # expanding out and and not
                = (if a                      # applying transformation above
                      (if nil a nil)
                    (if t a nil))
                = (if a                      # replacing a by t in THEN clause
                      (if nil t nil)
                    (if t a nil))
                = (if a                      # replacing a by nil in ELSE clause
                      (if nil t nil)
                    (if t nil nil))
                = (if a                      # (if nil t nil) -> nil
                      nil
                    (if t nil nil))
                = (if a nil nil)             # (if t nil nil) -> nil
```

When you're done all of this, you have an expression with only `if` expressions and with the property that for any "path" through the conditionals, you encounter every variable only once, in the condition of an `if`. A Boolean expression is a tautology exactly when every "path" through the conditionals lead to a value that is guaranteed to be non-nil.

If a branch has a return value that is not guaranteed to be non-nil (for instance, it is `nil`, or it is a variable), you can read off a counter example (that is, an assignment of truth values to variables that makes the Boolean expression false) by looking at the "path" through the conditionals that led to the values that is not guaranteed to be non-nil, setting a variable to `t` if you ended up going to the THEN branch of the conditional, or to `nil` if you ended up going to the right branch of the conditional.

Thus, in the above example, we ended up with `(if a nil nil)`; there are two "paths" into this conditional that lead to a value not guaranteed to be non-nil. One of those "path" is obtained by setting a to `t`, and the other is obtained by setting a to `nil`. It's easy to check that both of those assignments to a are counterexamples to the initial expression being a tautology.

For a more complex example, consider `(implies (or (not a) b) (or a (not b)))`.
If you work through the above steps, you get the simplified form `(if a (if b t t) (if b nil t))`. There is one "path" in the conditionals that lead to a value not guaranteed to be non-nil, and that's the "path" that leads to the `nil`, which is obtained by setting a to `nil` and setting b to `t`. And you can check that `a=nil,b=t` is a counterexample to the initial conjecture.

## Decision Procedure for Equivalences

This decision procedure is used to establish whether an implication where the hypotheses are equalities and the conclusion is also an equality is true, in the special case where this information can be decided without relying on the definition of the functions in the expression. (For instance, this may be interesting when we have *no* definition for the functions in the expression; that's what `defstub` does, by the way, it "defines" a function without actually giving an actual definition.)

Let's be more concrete. Suppose that we have no information about functions `f` and `g`, but we are asked to prove that:

```
(implies (and (= a c)
              (= (f a) b)
              (= (g c) c)
         (= (f (g a)) b))
```

Here's an argument that us humans can come up with easily, written in infix notation to simplify the presentation:

```
(f (g c)) = (f (g a))  # because a = c
          = (f c)      # because (g c) = c
          = (f a)      # because a = c
          = b          # because (f a) = b
```

No problem. How does ACL2s prove it, though? It uses a procedure called *congruence closure*. It has the advantage of working for an arbitrary equivalence relation, not just equality.

You can find precise definitions of the congruence closure algorithm in many places. See these slides on congruence closure if you are curious. I will describe it at a very high level here.

First off, take the formula you have, and extract all the subterms in it. The subterms are all the function applications that occur in the expression, and all the constants. List them all, the order is not important. Here are the subterms in the example above, listed in roughly in order of increasing complexity:

```
a
b
c
(f a)
(g c)
(g a)
(f (g a))
```

Note that `(g a)` is a subterm because it appears in `(f (g a))`.

Good. Now, the procedure is to repeatedly look at the equations in the hypotheses (there are three in our example), and "lump" together the terms in our list that the equation force to be equal. Lumping terms together may lead to further lumping, because new things may be made equal that we didn't know before were equal.

Let's start. Let's look at the first equation `a = c`. It tells you to lump together `a` and `c`. Let's do that:

```
a  c
b
(f a)
(g c)
(g a)
(f (g a))
```

I am representing lumps by putting terms on the same line. Terms lumped together are simply terms that can be proved to be equal using the equations we have been looking at. Knowing that `a` and `c` are equal tells you also that `(g a)` and `(g c)` are equal, so we can lump those two together too:

```
a  c
b
(f a)
(g c)  (g a)
(f (g a))
```

That's it, we cannot lump things together any further based on what we know.

Let's look at the second equation, `(f a) = b`. It tells you to lump together `(f a)` and `b`. Easy enough:

```
a  c
b  (f a)
(g c)  (g a)
(f (g a))
```

That's it, we cannot lump things together any further.

Let's look at the last equation, `(g c) = c`. It tells you to lump together `(g c)` and `c`. Now, because `c` is already lumped with something, and `(g c)` is also already lumped with something, let's merge the respective lumps. (After all, lumping just means that things can be proved equal.)

```
a  c  (g c)  (g a)
b  (f a)
(f (g a))
```

Oh, now we've learned a lot, in particular, that `(g a) = a`. This means that `(f (g a)) = (f a)`, so that we can lump `(f a)` and `(f (g a))` together:

```
a  c  (g c)  (g a)
b  (f a)  (f (g a))
```

And that's it, we cannot lump things any further. And we've exhausted all of our equations in the hypotheses. So the lumps above represent all the different things that can be proved equal to each other. Our conjecture asks whether the three equations imply that `(f (g a)) = b`; in other words, do the three equations ensure that `(f (g a))` and `b` are lumped together by the above procedure? It is easy to see that yes, they are. Therefore, the algorithm (and ACL2s) will report that the conjecture is true.

---

## April 21: Study Guide

As requested by a few students, here is a quick study guide for the final. (There is nothing here that has not been said already in class.) As we mentioned in class, the final will be a cross-section of all the tests we have had during the semester. Roughly, there should be one question per exam.

So what did those tests cover?

**Test 1:**

1. the basics of the ACL2 programming language
2. total functions,
3. the use of the design recipe to develop functions

**Test 2:**

1. lists and true lists
2. quotation
3. design recipe for recursive functions over true lists

**Test 3:**

1. Boolean logic
2. proving theorems using equational reasoning
3. falsifying conjectures

**Test 4:**

1. the induction principle (give proof obligations)
2. proving proof obligations

**Test 5:**

1. more difficult inductions (idenfitication of lemmas)
2. generalization
3. using different substitutions in the inductive proof obligation

**Test 6:**

1. The ACL2s theorem prover and the waterfall model
2. rewriting
3. understanding proof reports, including identifying interesting checkpoints
4. decision procedures for Boolean logic and congruence closure