Essential Logic for Computer Science
Answers to Even Problems

Rex Page and Ruben Gamboa

April 19, 2019

1.3. Deep Blue and Inductive Definitions

2. $factor(k, n) = (mod(n, k) = 0)$.

4. $p(n) = (lf(n) = 1)$.

2.1 Reasoning with Equations

2.

$$
\begin{array}{lll}
 & ((-1) \times x) + x & \\
= & (x \times (-1)) + x & \{ \times \text{ commutative } \} \\
= & (x \times (-1)) + (x \times 1) & \{ \times \text{ identity } \} \\
= & x \times ((-1) + 1) & \{ \text{ distributive law } \} \\
= & x \times 0 & \{ + \text{ complement } \} \\
= & 0 & \{ \times \text{ null } \}
\end{array}
$$

2.2 Boolean Equations

2.

$$
\begin{array}{lll}
 & (x \wedge y) \vee y & \\
= & y \vee (x \wedge y) & \{ \vee \text{ commutative } \} \\
= & (y \vee x) \wedge (y \vee y) & \{ \vee \text{ distributive } \} \\
= & (y \vee x) \wedge y & \{ \vee \text{ idempotent } \} \\
= & (x \vee y) \wedge y & \{ \vee \text{ commutative } \} \\
= & y & \{ \wedge \text{ absorption } \}
\end{array}
$$

2.3 Boolean Formulas

2. – Since $(x \wedge y) \vee y = y$, the truth table matches the value of y:

Formula	Value
$(True \wedge True) \vee True$	$True$
$(False \wedge True) \vee True$	$True$
$(True \wedge False) \vee False$	$False$
$(False \wedge True) \vee False$	$False$

– Since $(x \to y) \wedge (x \to (\neg y)) = (\neg x)$, the truth table matches the value of $\neg x$:

Formula	Value
$(True \to True) \wedge (True \to (\neg True))$	$False$
$(True \to False) \wedge (True \to (\neg False))$	$False$
$(False \to True) \wedge (False \to (\neg True))$	$True$
$(False \to False) \wedge (False \to (\neg False))$	$True$

– The formula is not grammatically correct, so it has no truth table.

4.

$$
\begin{array}{lll}
 & (x \to y) \wedge (y \to x) & \\
= & (x \to y) \wedge ((\neg x) \to (\neg y)) & \{ \text{ contrapositive } \}
\end{array}
$$

6.

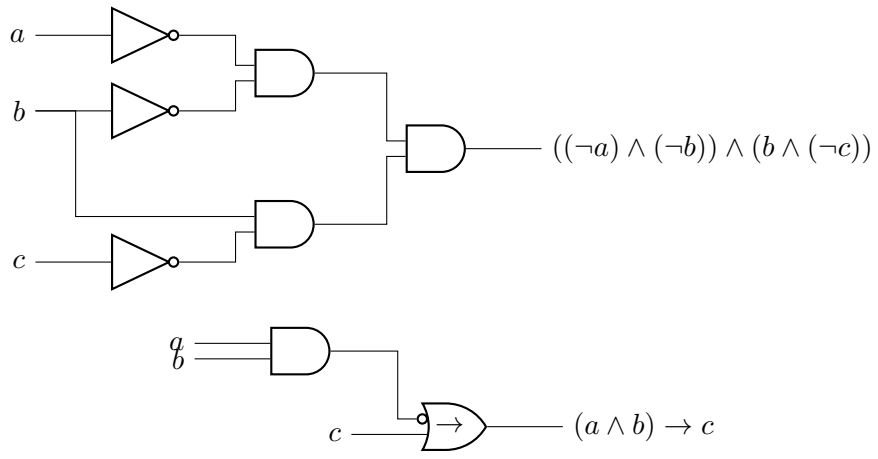
$$
\begin{array}{lll}
 & \neg((x \wedge (\neg y)) \vee ((\neg x) \wedge y)) & \\
= & (\neg(x \wedge (\neg y))) \wedge (\neg((\neg x) \wedge y)) & \{ \vee \text{ deMorgan } \} \\
= & ((\neg x) \vee (\neg(\neg y))) \wedge ((\neg(\neg x)) \vee (\neg y)) & \{ \wedge \text{ deMorgan } \} \text{ twice} \\
= & ((\neg x) \vee y) \wedge (x \vee (\neg y)) & \{ \text{ double negation } \} \text{ twice} \\
= & ((\neg x) \vee y) \wedge ((\neg y) \vee x) & \{ \vee \text{ commutative } \} \\
= & (x \to y) \wedge (y \to x) & \{ \text{ implication } \} \text{ twice}
\end{array}
$$

8.

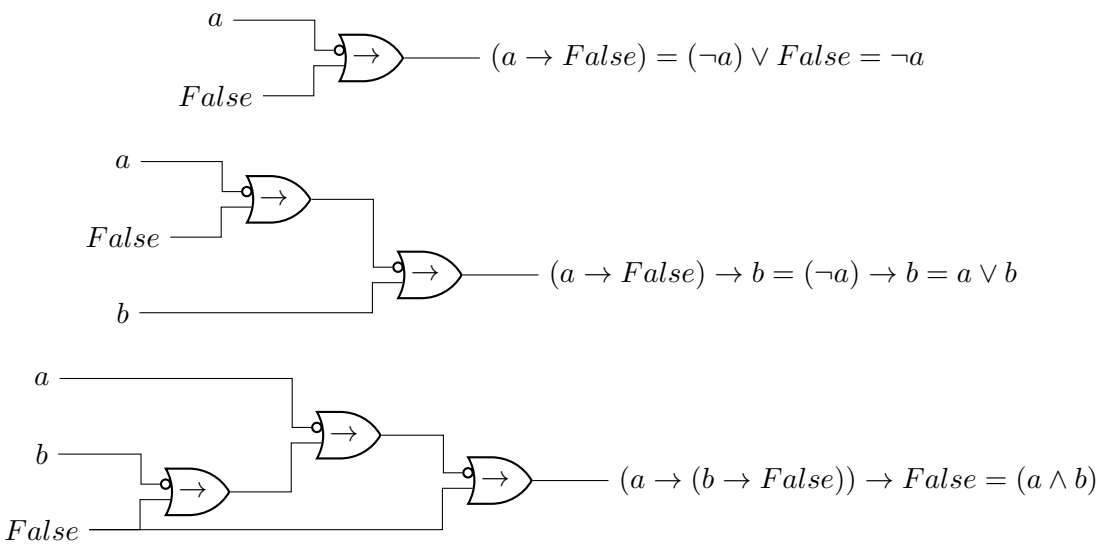
$$
\begin{array}{lll}
 & (x \wedge y) \vee ((\neg x) \wedge (\neg y)) & \\
= & ((x \wedge y) \vee (\neg x)) \wedge ((x \wedge y) \vee (\neg y)) & \{ \vee \text{ distributive } \} \\
= & ((\neg x) \vee (x \wedge y)) \wedge ((\neg y) \vee (x \wedge y)) & \{ \vee \text{ commutative } \} \\
= & (((\neg x) \vee x) \wedge ((\neg x) \wedge y)) \wedge (((\neg y) \vee x) \wedge ((\neg y) \vee y)) & \{ \vee \text{ distributive } \} \\
= & ((x \to x) \wedge (x \to y)) \wedge ((y \to x) \wedge (y \to y)) & \{ \text{ implication } \} \\
= & (True \wedge (x \to y)) \wedge ((y \to x) \wedge True) & \{ \text{ self implication } \} \\
= & ((x \to y) \wedge True) \wedge ((y \to x) \wedge True) & \{ \textit{wedge} \text{ commutative } \} \\
= & (x \to y) \wedge (y \to x) & \{ \textit{wedge} \text{ identity } \}
\end{array}
$$

2.4 Digital Circuits

2.



4.



2.5 Deduction

2.

Assume (a)
- - - - - - - - - - - - - - - - - - -
Assume $(a \to b)$
Assume $(b \to c)$
———————————{$\to$ chain}
$a \to c$
———————————{modus ponens}
c

4. The theorem "$a, a \to b, b \to c \vdash c$" is equivalent to the Boolean equation $((a \wedge (a \to b) \wedge (b \to c)) \to c) = True$.

6.

Assume $(\neg(\neg(b \vee a)))$
———————————{$\neg\neg$ fwd}
$b \vee a$
———————————{$\vee$ commutes}
$a \vee b$
- - - - - - - - - - - - - - - - - - -
Assume $(\neg(a \vee b))$
———————————{$\neg$ elim}
$(a \vee b) \to False$
———————————{modus ponens}
$False$
———————————{$\to$ intro}
$(\neg(\neg(b \vee a))) \to False$
———————————{reductio ad absurdum}
$(\neg(b \vee a))$

8.

Assume (a)
———————————{$\vee$ intro 1}
$a \vee b$
- - - - - - - - - - - - - - - - - -
Assume $(\neg(a \vee b))$
———————————{$\neg$ elim}
$(a \vee b) \rightarrow False$
———————————{modus ponens}
$False$
———————————{$\rightarrow$ intro}
$a \rightarrow False$
———————————{$\neg$ intro}
$\neg a$

10.

Assume $(a \vee b)$
- - - - - - - - - - - - - - - - - -
Assume (a)
- - - - - - - - - - - - - - - - - -
Assume $((\neg a) \wedge (\neg b))$
———————————{$\wedge$ elim 1}
$\neg a$
———————————{$\neg$ elim}
$a \rightarrow False$
- - - - - - - - - - - - - - - - - -
Assume (b)
- - - - - - - - - - - - - - - - - -
Assume $((\neg a) \wedge (\neg b))$
———————————{$\wedge$ elim 2}
$\neg b$
———————————{$\neg$ elim}
$b \rightarrow False$
———————————{modus ponens}
$False$
———————————{$\vee$ elim}
$(a \vee b) \rightarrow False$
———————————{$\neg$ intro}
$\neg(a \vee b)$

12.

Assume $(\neg(a \vee (\neg a)))$
———————{nor commutes}
$(\neg((\neg a) \vee a))$
———————{nor elim 1}
$(\neg(\neg a))$
———————{¬¬ fwd}
a
- - - - - - - - - - - - - - - - - -
Assume $(\neg(a \vee (\neg a)))$
———————{nor elim 1}
$\neg a$
———————{¬ elim}
$a \rightarrow False$
———————{modus ponens}
$False$
———————{→ intro}
$(\neg(a \vee (\neg a))) \rightarrow False$
———————{reductio ad absurdum}
$(a \vee (\neg a))$

2.6 Predicates and Quantifiers

2.

b) No free variables. Two variables bound (x and y).

c) One free variable (x). One variable bound (y).

e) One free variable (y). Two variables bound (x and the other x).

2.7 Reasoning with Quantified Predicates

2. Since the universe of discourse is empty, $(\forall x.P(x))$ is $True$, and $(\exists x.P(x))$ is $False$. So, $(\forall x.P(x)) \rightarrow (\exists x.P(x)) = True \rightarrow False = False$.

4. This can be done in two parts. First, reduce quantifier reasoning to propositional reasoning:

$$
\begin{array}{ll}
 & ((\forall x.(P(x) \rightarrow Q(x))) \wedge (\forall x.(Q(x) \rightarrow R(x)))) \rightarrow (\forall x.(P(x) \rightarrow R(x))) \\
\Leftarrow & ((\forall x.(P(x) \rightarrow Q(x))) \wedge (\forall y.(Q(y) \rightarrow R(y)))) \rightarrow (\forall z.(P(z) \rightarrow R(z))) \\
 & \qquad \{\text{Renaming}\} \\
\Leftarrow & (\forall z.(\exists x.(\exists y.((P(x) \rightarrow Q(x)) \wedge (Q(y) \rightarrow R(y))) \rightarrow (P(z) \rightarrow R(z))))) \\
 & \qquad \{\text{Migrating}\} \\
\Leftarrow & ((P(z_0) \rightarrow Q(z_0)) \wedge (Q(z_0) \rightarrow R(z_0))) \rightarrow (P(z_0) \rightarrow R(z_0)) \\
 & \qquad \{\text{Removing}\} \\
= & True \\
 & \qquad \{\text{Propositional Reasoning}\}
\end{array}
$$

Then, do the necessary propositional reasoning: Prove that $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$. Note: $(x \wedge y) \vee z = ((x \vee z) \wedge (y \vee z)$ {$\vee$ distributive R} is an equation that is easily proved, citing {$\vee$ commutative} and {$\vee$ distributive}.

$$
\begin{array}{lll}
 & ((p \to q) \wedge (q \to r)) \to (p \to r) & \\
= & \neg((p \to q) \wedge (q \to r)) \vee (\neg p \vee r) & \{\text{implication}\} \text{ twice} \\
= & (\neg(p \to q) \vee \neg(q \to r)) \vee (\neg p \vee r) & \{\text{De Morgan}\} \\
= & ((p \wedge \neg q) \vee (q \wedge \neg r)) \vee (\neg p \vee r) & \{\to \text{ negation}\} \text{ twice} \\
= & (((p \wedge \neg q) \vee q) \wedge ((p \wedge \neg q) \vee \neg r)) \vee (\neg p \vee r) & \{\vee \text{ distributive}\} \\
= & ((p \vee q) \wedge (\neg q \vee q) \wedge (p \vee \neg r) \wedge (\neg q \vee \neg r)) \vee (\neg p \vee r) & \\
 & & \{\vee \text{ distributive R}\} \text{ twice} \\
= & ((p \vee q) \vee (\neg p \vee r)) \wedge ((\neg q \vee q) \vee (\neg p \vee r)) \wedge ((p \vee \neg r) \vee (\neg p \vee r)) \wedge ((\neg q \vee \neg r) \vee (\neg p \vee r)) & \\
 & & \{\wedge \text{ associative}\}, \{\vee \text{ distributive R }\} \text{ 4 times} \\
= & ((\neg p \vee p) \vee (q \vee r)) \wedge ((\neg q \vee q) \vee (\neg p \vee r)) \wedge ((\neg r \vee r) \vee (\neg p \vee p)) \wedge ((\neg r \vee r) \vee (\neg q \vee \neg p)) & \\
 & & \{\wedge \text{ associative}\}, \{\vee \text{ commutative}\} \text{ 4 times} \\
= & (True \vee (q \vee r)) \wedge (True \vee (\neg p \vee r)) \wedge (True \vee (\neg p \vee p)) \wedge (True \vee (\neg q \vee \neg p)) & \\
 & & \{\text{excluded middle}\} \text{ 4 times} \\
= & True \wedge True \wedge True \wedge True & \{\vee \text{ null}\} \text{ 4 times} \\
= & True & \{\wedge \text{ associative}\} \text{ twice}, \{\wedge \text{ idempotent}\} \text{ twice}
\end{array}
$$

6. Suppose $((\exists x.P(\dots x \dots)) \wedge Q) = True$. Then, by the truth table of $\wedge$, $(\exists x.P(\dots x \dots)) = True$ and $Q = True$. By definition of $\exists$, there is a value x_0 such that, $P(\dots x_0 \dots) = True$, so $P(\dots x_0 \dots) \wedge Q = True$. Then by the definition of $\exists$, $(\exists x.(P(\dots x \dots) \wedge Q)) = True$.

3. Software Testing and Prefix Notation

2.
```
(defproperty max>=both
   (x :value (random-integer)
    y :value (random-integer))
   (and (>= (max x y) x)
        (>= (max x y) y)))
```

4.
```
(defproperty mod-consistent
   (m-minus-1 :value (random-natural)
    x         :value (random-integer)
    y         :value (random-integer))
   (let* ((m (+ m-minus-1 1))) ; avoid zero modulus
     (and (= (mod (+ x y) m)
             (mod (+ (mod x m) (mod y m)) m))
          (= (mod (- x y) m)
             (mod (- (mod x m) (mod y m)) m))
          (= (mod (* x y) m)
             (mod (* (mod x m) (mod y m)) m)))))
```

4.2. Mathematical Induction

2. $E(n) \equiv (\mathsf{expt}\ x\ n) = x^n$.

   Base case, $E(0)$:

$$
\begin{array}{lll}
(\mathsf{expt}\ x\ 0) & = 1 & \{\text{expt0}\} \\
 & = x^0 & \{\text{algebra}\}
\end{array}
$$

Induction case, $E(n) \rightarrow E(n+1)$:

$$\begin{array}{lll} (\mathsf{expt}\ x\ n+1) & = (\mathsf{*}\ x\ (\mathsf{expt}\ x\ n)) & \{\text{expt1}\} \\ & = (\mathsf{*}\ x\ x^n) & \{E(n)\} \\ & = x^{n+1} & \{\text{algebra}\} \end{array}$$

4.4. Concatenation, Prefixes, and Suffixes

2. $R(n) \equiv ((\mathsf{rest}\ [x_1\ x_2 \ldots x_n]) = (\mathsf{nthcdr}\ 1\ [x_1\ x_2 \ldots x_n]))$

   Zero case: $R(0)$

$$\begin{array}{lll} (\mathsf{rest}\ \mathsf{nil}) & = \mathsf{nil} & \{\text{rst0}\} \\ & = (\mathsf{nthcdr}\ 1\ \mathsf{nil}) & \{\text{sfx0}\} \end{array}$$

   Non-zero case: $R(n+1)$

$$\begin{array}{lll} (\mathsf{rest}\ [x_1\ x_2 \ldots x_{n+1}]) & = (\mathsf{rest}\ (\mathsf{cons}\ x_1\ [x_2 \ldots x_{n+1}])) & \{\text{cons}\} \\ & = [x_2 \ldots x_{n+1}] & \{\text{rst1}\} \\ & = (\mathsf{nthcdr}\ 0\ [x_2 \ldots x_{n+1}]) & \{\text{sfx0}\} \\ & = (\mathsf{nthcdr}\ (+\ 0\ 1)\ (\mathsf{cons}\ x_1\ [x_2 \ldots x_{n+1}])) & \{\text{sfx1}\} \\ & = (\mathsf{nthcdr}\ 1\ (\mathsf{cons}\ x_1\ [x_2 \ldots x_{n+1}])) & \{\text{arithmetic}\} \\ & = (\mathsf{nthcdr}\ 1\ [x_1\ x_2 \ldots x_{n+1}]) & \{\text{cons}\} \end{array}$$

   Note: This is a case-by-case proof. No induction needed.

4. $A(n) \equiv ([x_1\ x_2 \ldots x_n] = (\mathsf{append}\ [x_1\ x_2 \ldots x_n]\ \mathsf{nil}))$

   Base case: $A(0)$

$$\mathsf{nil} \quad = (\mathsf{append}\ \mathsf{nil}\ \mathsf{nil}) \quad \{\text{app0}\}$$

   Inductive case, $A(n) \rightarrow A(n+1)$:

$$\begin{array}{lll} [x_1\ x_2 \ldots x_{n+1}] & = (\mathsf{cons}\ x_1\ [x_2 \ldots x_{n+1}]) & \{\text{cons}\} \\ & = (\mathsf{cons}\ x_1\ (\mathsf{append}\ [x_2 \ldots x_{n+1}]\ \mathsf{nil})) & \{A(n)\} \\ & = (\mathsf{append}\ (\mathsf{cons}\ x_1\ [x_2 \ldots x_{n+1}])\ \mathsf{nil}) & \{\text{app1}\} \\ & = (\mathsf{append}\ [x_1\ x_2 \ldots x_{n+1}])\ \mathsf{nil}) & \{\text{cons}\} \end{array}$$

6. $M(n) \equiv (\ (\mathsf{member\text{-}equal}\ y\ (\mathsf{rep}\ n\ x)) \rightarrow (\mathsf{member\text{-}equal}\ y\ (\mathsf{cons}\ x\ \mathsf{nil}))\ )$

   Base case: $M(0)$

$$\begin{array}{lll} & (\mathsf{member\text{-}equal}\ y\ (\mathsf{rep}\ 0\ x)) \rightarrow (\mathsf{member\text{-}equal}\ y\ (\mathsf{cons}\ x\ \mathsf{nil})) & \\ = & (\mathsf{member\text{-}equal}\ y\ \mathsf{nil}) \rightarrow (\mathsf{member\text{-}equal}\ y\ (\mathsf{cons}\ x\ \mathsf{nil})) & \{\text{rep0}\} \\ = & \mathsf{nil} \rightarrow (\mathsf{member\text{-}equal}\ y\ (\mathsf{cons}\ x\ \mathsf{nil})) & \{\text{mem0}\} \\ = & True & \{\rightarrow \text{truth table}\} \end{array}$$

   Inductive case: $M(n) \rightarrow M(n+1)$

   Lemma: $((x \vee y) \rightarrow z) = ((x \rightarrow z) \wedge (y \rightarrow z))$.

$$\begin{array}{lll} ((x \vee y) \rightarrow z) & = (\neg(x \vee y) \vee z) & \{\text{ implication }\} \\ & = ((\neg x) \wedge (\neg y)) \vee z & \{\ \vee \text{ DeMorgan }\} \\ & = ((\neg x) \vee z) \wedge ((\neg y) \vee z) & \{\ \vee \text{ DeMorgan}\},\ \{\vee \text{ commutative }\} \\ & = (x \rightarrow z) \wedge (y \rightarrow z) & \{\text{ implication }\} \end{array}$$

$$
\begin{array}{lll}
 & \text{(member-equal } y \text{ (rep (+ } n \text{ 1) } x)) \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))} & \\
= & \text{(member-equal } y \text{ (cons } x \text{ (rep } n \ x))) & \\
 & \quad \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))} & \{\text{rep1}\} \\
= & ((\text{equal } y \ x) \vee \text{(member-equal } y \text{ (rep } n \ x))) & \\
 & \quad \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))} & \{\text{mem1}\} \\
= & ((\text{equal } y \ x) \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))}) \wedge & \\
 & \quad ((\text{member-equal } y \text{ (rep } n \ x)) \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))}) & \{\text{Lemma}\} \\
= & ((\text{equal } y \ x) \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))}) \wedge True & \{M(n)\} \\
= & \text{(equal } y \ x) \rightarrow \text{(member-equal } y \text{ (cons } x \text{ nil))} & \{\wedge \text{ identity}\} \\
= & \text{(equal } y \ x) \rightarrow ((\text{equal } y \ x) \vee \text{(member-equal } y \text{ nil)}) & \{\text{mem1}\} \\
= & \text{(equal } y \ x) \rightarrow ((\text{equal } y \ x) \vee False) & \{\text{mem0}\} \\
= & \text{(equal } y \ x) \rightarrow \text{(equal } y \ x) & \{\vee \text{ identity}\} \\
= & True & \{\text{self-implication}\}
\end{array}
$$

5.3. Theorems with Constraints

2.
```
(include-book "arithmetic-3/top" :dir :system)
(defthm nthcdr-zap-thm
  (implies (true-listp xs)
           (equal (nthcdr (len xs) xs)
                  nil)))
```

6.1. Numbers and Numerals

2. Note: a and b are non-zero, natural numbers. Otherwise, (* a b) would violate a restriction on the second operand of the mod operator, making (mod (* a x) (* a b)) an invalid formula.

   Let $r_1 \equiv$ (mod x b), and $q_1 \equiv$ (floor x b). Then $x = q_1 b + r_1$. Similarly, let $r_2 \equiv$ (mod (* a x) (* a b)), and $q_2 \equiv$ (floor (* a x) (* a b)). Then (* a x) $= q_2 \cdot$ (* a b) $+ r_2$.

   Observe that $q_2 = q_1$.

$$
\begin{array}{lll}
q_2 & = \left\lfloor \frac{(* \ a \ x)}{(* \ a \ b)} \right\rfloor & \{\text{defn floor}\} \\
 & = \left\lfloor \frac{a}{b} \right\rfloor & \{\text{algebra}\} \\
 & = \text{(floor } a \ b) & \{\text{defn floor}\} \\
 & = q_1 & \{\text{defn } q_1\}
\end{array}
$$

   The rest follows from algebra.

$$
\begin{array}{lll}
 & q_1 \cdot (* \ a \ b) + r_2 & \\
= & q_2 \cdot (* \ a \ b) + r_2 & \{q_2 = q_1\} \\
= & (* \ a \ b) & \{\text{check remainder}\} \\
= & (* \ a \ (q_1 b + r_1)) & \{\text{check remainder}\} \\
= & q_1 \cdot (* \ a \ b) + (* \ a \ r_1) & \{\text{algebra}\}
\end{array}
$$

   Subtracting $q_1 \cdot (* \ a \ b)$ from both sides, we get that $r_2 = (* \ a \ r_1)$.
   That is, (mod (* a x) (* a b)) = (* a (mod x b)).

6.2. Numbers from Numerals

2.
```
(defproperty numb10-inverts-dgts
```

```
(n :value (random-natural))
(= (numb10 (dgts n)) n))
```

4. (dgts (numb10 [0])) = [ ] ≠ [0]. Note that (dgts (numb10 [ ])) = [ ].

6. $M(n) \equiv (x_n > 0) \rightarrow ((\text{dgts (numb10 } [x_0\ x_1 \ldots x_n])) = [x_0\ x_1 \ldots x_n])$

   Base case, $M(0)$. Assume $x_n > 0$ and prove $((\text{dgts (numb10 } [x_0\ x_1 \ldots x_n])) = [x_0\ x_1 \ldots x_n])$.

	(dgts (numb10 [x_0]))	
=	(dgts (+ x_0 (* 10 (numb10 nil))))	{numb10.1}
=	(dgts (+ x_0 (* 10 0)))	{numb10.0}
=	(dgts x_0)	{algebra}
=	(cons (mod x_0 10) (dgts (floor x_0 10)))	{dgts1}, since $x_0 > 0$
=	(cons x_0 (dgts 0))	{arithmetic}
=	(cons x_0 nil)	{dgts0}
=	[x_0]	{cons}

   Inductive case, $M(n) \rightarrow M(n+1)$. Again assume $x_{n+1} > 0$.

	(dgts (numb10 [$x_0\ x_1 \ldots x_{n+1}$]))	
=	(dgts (+ x_0 (* 10 (numb10 [$x_1\ \ldots\ x_{n+1}$]))))	{numb10.1}
=	(cons (mod (+ x_0 (* 10 (numb10 [$x_1\ \ldots\ x_{n+1}$]))) 10) (dgts (floor (+ x_0 (* 10 (numb10 [$x_1\ \ldots\ x_{n+1}$]))) 10)))	{dgts1}, since $x_0 > 0$
=	(cons x_0 (dgts (numb10 [$x_1\ \ldots\ x_{n+1}$])))	{mod}, {floor} since $x_0 < 10$
=	(cons x_0 [$x_1\ \ldots\ x_{n+1}$])	{$M(n)$}
=	[$x_0\ x_1 \ldots x_{n+1}$]	{cons}

6.3. Binary Numerals

2. $B(n) \equiv (\text{numb (bits } n)) = n$

   Base case, $B(0)$

	(numb (bits 0))	
=	(numb nil)	{ bits0 }
=	0	{ numb0 }

   Inductive case (strong induction), $(\forall m < n+1.B(m)) \rightarrow B(n+1)$

	(numb (bits (+ n 1)))	
=	(numb (cons (mod (+ n 1) 2) (bits (floor (+ n 1) 2))))	{ bits1 }
=	(+ (mod (+ n 1) 2) (* 2 (numb (bits (floor (+ n 1) 2)))))	{ numb1 }
=	(+ (mod (+ n 1) 2) (* 2 (floor (+ n 1) 2)))	{ B(floor (+ n 1) 2) } Note: (floor (+ n 1) 2) < $n+1$
=	(+ n 1)	{ check remainder }

4. First, be sure to include the arithmetic libraries.

   ```
   (include-book "arithmetic-3/floor-mod/floor-mod" :dir :system)
   ```

   Then, enter the definitions of numb and bits, along with the specified theorem.

6. $F(n) \equiv (\mathsf{fin}\ (\mathsf{bits}\ (+\ n\ 1))) = 1$.

Base case, $F(0)$.

	(fin (bits (+ 0 1)))	
=	(fin (bits 1))	{ arithmetic }
=	(fin (cons (mod 1 2) (bits (floor 1 2))))	{ bits1 }
=	(fin (cons 1 (bits 0)))	{ defns mod, floor }
=	(fin (cons 1 nil))	{ bits0 }
=	(first (cons 1 nil))	{ fin1 }
=	1	{ fst }

Inductive case (strong induction), $(\forall m < n+1.F(m)) \rightarrow F(n+1)$

	(fin (bits (+ $(n+1)$ 1)))	
=	(fin (cons (mod (+ $(n+1)$ 1) 2) (bits (floor (+ $(n+1)$ 1) 2))))	{ bits1 }
=	(fin (cons (mod (+ $(n+1)$ 1) 2) (bits (+ (floor n 2) 1))))	{ algebra}
=	(fin (bits (+ (floor n 2) 1)))	{ fin2 }, since (+ (floor n 2) 1) > 0
=	1	{ F(+ (floor n 2) 1) }
		Note: (+ (floor n 2) 1) $<$ (+ $(n+1)$ 1)

8. $L(n) \equiv ((\lfloor 2^{n-1} \rfloor \leq m < 2^n) \rightarrow ((\mathsf{len}\ (\mathsf{bits}\ m)) = n))$

Base case: $L(0)$ (Note: $0 \leq m < 2^0$, so $m = 0$)

	(len (bits 0))	
=	(len nil)	{ bits0 }
=	0	{ len0 }

Inductive case: $L(n) \rightarrow L(n+1)$
Assume $\lfloor 2^n \rfloor \leq m < 2^{n+1}$. Prove (len (bits m)) $= n+1$.

Algebraic observations:
$m > 0$ {alg1}
$\lfloor 2^{n-1} \rfloor \leq$ (floor m 2) $< 2^n$ {alg2}

	(len (bits m))	
=	(len (cons (mod m 2) (bits (floor m 2))))	{alg1}, { bits1 }
=	(+ 1 (len (bits (floor m 2))))	{ len1 }
=	(+ 1 n)	{alg2}, { $L(n)$ }
=	$n+1$	{algebra}

10. Note: $2^{\lfloor \log_2(n) \rfloor} \leq n < 2^{\lfloor \log_2(n) \rfloor + 1}$. So by Exercise 8, (len (bits n)) $= \lfloor \log_2(n) \rfloor + 1$.

12. $R(m) \equiv (\mathsf{numb}\ (\mathsf{append}\ (\mathsf{bits}\ n)\ (\mathsf{rep}\ m\ 0))) = (\mathsf{numb}\ (\mathsf{bits}\ n))$.

Base case: $R(0)$.

	(numb (append (bits n) (rep 0 0)))	
=	(numb (append (bits n) nil))	{ rep0 }
=	(numb (bits n))	{ app-nil }

Inductive case: $R(m) \rightarrow R(m+1)$.

	(numb (append (bits n) (rep (+ 1 m) 0)))	
=	(numb (append (bits n) (cons 0 (rep m 0)))	{ rep1 }
=	(numb (append (bits n) (append (cons 0 nil) (rep m 0)))	{ app1, app0 }
=	(numb (append (bits n) (append (list 0) (rep m 0)))	{ defn list }
=	(numb (append (append (bits n) (list 0)) (rep m 0)))	{ app associative }
=	(numb (append (bits n) (list 0)))	{ $R(m)$ }
=	(numb (bits n)))	{ leading-0 } (Exercise 11)

7.1. Adding Numerals

2.

		1	0	1	1	multiplicand
×			1	0	1	multiplier
		1	0	1	1	2^0's bit of multiplier times multiplicand
	0	0	0	0		2^1's bit of multiplier times multiplicand, shifted 1 place left
1	0	1	1			2^2's bit of multiplier times multiplicand, shifted 2 places left
1	1	0	1	1	1	sum of numerals and shifted numerals ($11 \times 5 = 55$)

7.3. Circuits for Adding Two-Bit Binary Numerals

2. A really good answer requires advanced analysis based on probability theory. However, less advanced methods can help size up the situation.

   If we generate five random bits, the likelihood that all are zeros is $(1/2)^5$, which is $1/32$. That means the probability of not getting all zeros is $31/32$. If we do n tests, the probability of not hitting the all-zeros case at least once is $1 - (31/32)^n$.

   Since we want to be confident that the all-zeros case is tested at least once, we should at least do enough tests to make it highly probable that all five bits are zero in at least one test. With 150 tests, the probability that the all zeros case never comes up is a little over 99%: $1 - (31/32)^{150} > 99\%$.

   That is, 150 tests would make us confident of testing the all-zeros case at least once. What about the case where the first four bits are zeros and the fifth is a 1? That's a 1 in 32 longshot, too, as is any other particular pattern of five bits. Maybe all of those bit patterns would come up in 150 tests, or maybe not. A formula for the probability is too complicated to discuss here, but it is surely less likely that we will hit all bit patterns at least once in n tries than that we will get the all-zeros pattern at least once. So, 150 tries is probably not going to be enough to have 99% confidence of testing all cases as least once, but 1,000 tests might be a good start.

7.4. Adding w-Bit Binary Numerals

2.
```
(defthm adder-ok
   (let* ((a (adder c0 x y))
          (s (first a)) (c (second a)))
         (= (numb (append s (list c)))
```

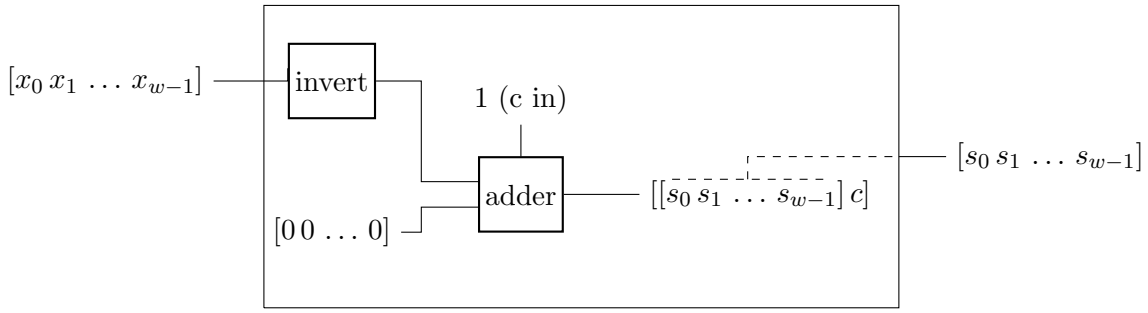
```
(+ (numb (list c0)) (numb x) (numb y)))))
```

7.5. Numerals for Negative Numbers

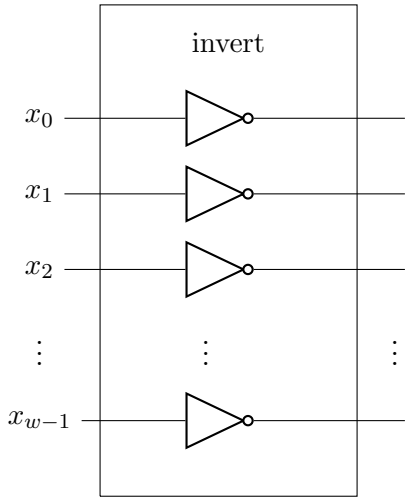
2. Assume $-2^{w-1} \leq n < 0$.

	(fin (twos w n))	
=	(fin (bits (+ (expt 2 w) n)))	{ defun twos, $n < 0$ }
=	1	{ hi-1 Exercise 6, page 136 } since $2^w + n \geq 0$

4.



The "invert" circuit flips each incoming wire by using a separate NOT gate for each wire:



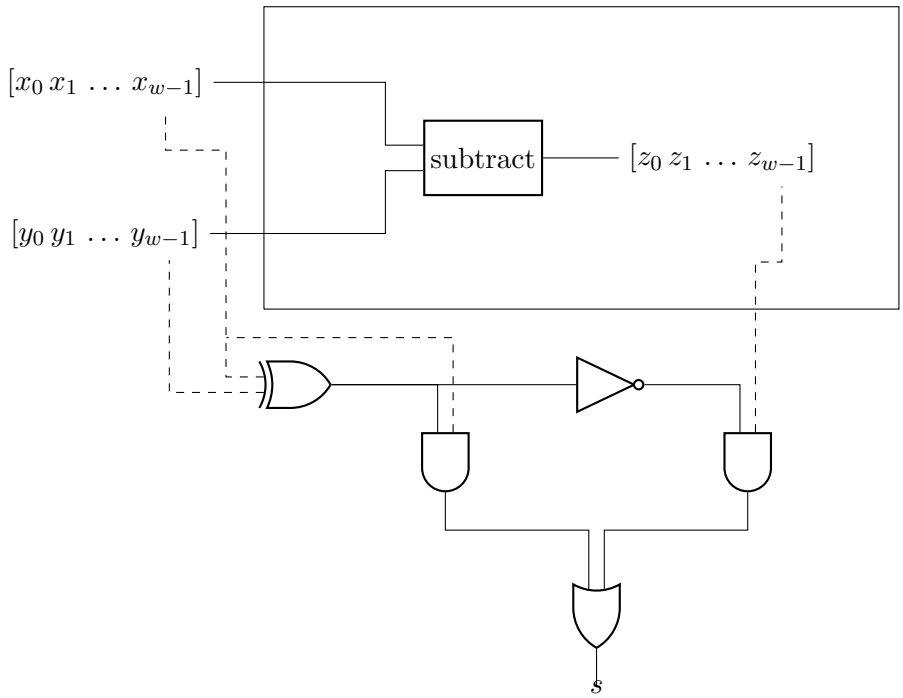
6.
```
(defun invert-bits (xs)
   (if (consp xs)
       (cons (- 1 (first xs))
             (invert-bits (rest xs)))
       nil))
(defproperty negate-test
   (n :value (random-between 1 (expt 2 31)))
   (let* ((xs (pad 32 0 (bits n)))
          (ys (invert-bits xs)))
      (equal (bits (+ 1 (numb ys)))
             (twos 32 (- n)))))
```

8.
```
(defun subtract (xs ys)
   (let* ((ss-c (adder 0 (pad (len xs) 0 (list 1)) (invert-bits ys)))
          (neg-ys (first ss-c))) ; ignore carry-bit
      (first (adder 0 xs neg-ys))))

; or this (perhaps too clever by half):
(defun tricky-subtract (xs ys)
   (first (adder 1 xs (invert-bits ys))))
```

10. When the numerals to be compared have the same sign, the difference the difference between the numbers they represent will be in the range of numbers that a w-bit, two's-complement numeral can represent, namely numbers in the range from -2^{w-1} to $2^{w-1}-1$. So, the numerals can be compared by subtracting one from the other and checking the sign of the result.

    However, when the numerals to be compared have different signs, the difference can be outside the representable range. For example, if one input numeral represents $2^{w-1}-1$ and the other represents -2^{w-1}, the difference would be $2^{w-1}-1-(-2^{w-1}) = 2^w - 1$, which is outside the representable range for w-bit, two's-complement numerals. Fortunately, when the signs are different, the input with a minus-sign represents the smaller number, so the comparison circuit only needs to check signs. No subtraction is required. The following circuit makes use of this observation.



11.
```
(defun and-gate (x y) (if (and (= x 1) (= y 1)) 1 0))
(defun or-gate (x y) (if (or (= x 1) (= y 1)) 1 0))
(defun xor-gate (x y) (if (and (= x 1) (= y 1)) 0 (or-gate x y)))
(defun not-gate (x) (if (= x 1) 0 1))
(defun less (xs ys) ; xs, ys are twos comp numerals with same word size
```

```
(let* ((x-sign (fin xs))
       (y-sign (fin ys))
       (zs     (subtract xs ys))
       (z-sign (fin zs))
       (xy-signs-different (xor-gate x-sign y-sign)))
      (or-gate (and-gate z-sign (not-gate xy-signs-different))
               (and-gate x-sign xy-signs-different))))
```

12.

```
(defproperty comparison-test
   (w-2 :value (random-between 0 32) ; word-size 2 or more, not too big
    x   :value (random-between (- (expt 2 (+ w-2 1)))
                               (expt 2 (- (+ w-2 1) 1)))
    y   :value (random-between (- (expt 2 (+ w-2 1)))
                               (expt 2 (- (+ w-2 1) 1))))
   (let* ((w (+ w-2 2)))
     (iff (< x y)
          (= 1 (less (twos w x) (twos w y))))))
```

8.1. Bignum Adder

12.

```
(defthm no-leading-zeros
   (implies (and (consp x) (consp y)
                 (= (fin x) 1) (= (fin y) 1))
            (= (fin (add 0 x y)) 1)))
```

9.1. Multiplexer

4. Prove, separately, the following two cases:

   – Case 0x (1st operand empty): (mux2 nil ys) = (mux nil ys)

   – Case 1x (1st operand not empty): (mux2 (cons x xs) ys) = (mux (cons x xs) ys)

   The proofs follow.

   – Case 0x:

	(mux2 nil ys)	
=	ys	{mux2-0x}
=	(mux nil ys)	{mux0x}

   – Case 1x: Proof by induction.

   $M(n) \equiv$ ((mux2 (cons x xs) [y_1 y_2 ... y_n]) = (mux (cons x xs) [y_1 y_2 ... y_n]))

   * Base case: $M(0)$.

(mux2 (cons x xs) nil)
= (mux2 (cons x xs) nil) {cons}
= (cons (first(cons x xs)) (mux2 nil (rest(cons x xs)))) {mux2-1x}
= (cons (first(cons x xs)) (rest(cons x xs))) {mux2-0x}
= (cons x xs) {fst}, {rst}
= (mux (cons x xs) nil) {mux0y}

* Inductive case: $M(n) \rightarrow M(n+1)$.

(mux2 (cons x xs) [y_1 y_2 ... y_{n+1}])
= (cons (first(cons x xs)) (mux2 [y_1 y_2 ... y_{n+1}] (rest(cons x xs)))) {mux2-1x}
= (cons x (mux2 [y_1 y_2 ... y_{n+1}] xs)) {fst}, {rst}
= (cons x (cons (first[y_1 y_2 ... y_{n+1}]) (mux2 xs (rest[y_1 y_2 ... y_{n+1}])))) {mux2-1x}
= (cons x (cons y_1 (mux2 xs [y_2 ... y_{n+1}]))) {fst}, {rst}
= (cons x (cons y_1 (mux xs [y_2 ... y_{n+1}]))) $M(n)$ or {Case 0x}
= (mux (cons x xs) (cons y_1 [y_2 ... y_{n+1}])) mux11
= (mux (cons x xs) [y_1 y_2 ... y_{n+1}]) cons

9.2. Demultiplexer

6. Proofs by induction separately for operands with an even versus odd number of elements.

   – Even case: $E(n) \equiv$ ((mux (first (dmx [x_1 y_1 x_2 y_2 ... x_n y_n])) (second (dmx [x_1 y_1 x_2 y_2 ... x_n y_n])))) = [x_1 y_1 x_2 y_2 ... x_n y_n])

   – Odd case: $D(n) \equiv$ ((mux (first (dmx [x_1 y_1 x_2 y_2 ... x_n y_n x_{n+1}])) (second (dmx [x_1 ys x_2 y_2 ... x_n y_n x_{n+1}])))) = [x_1 y_1 x_2 y_2 ... x_n y_n x_{n+1}])

   The proofs follow.

   – Even base case: $E(0)$.

(mux (first (dmx nil)) (second (dmx nil)))
= (mux (first [nil nil]) (second [nil nil])) {dmx0} twice
= (mux nil nil) {fst}, {rst}
= nil {mux0x}

   – Even inductive case: $E(n) \rightarrow E(n+1)$.

(mux (first (dmx [x_1 y_1 x_2 y_2 ... x_{n+1} y_{n+1}])) (second(dmx [x_1 y_1 x_2 y_2 ... x_{n+1} y_{n+1}])))
= (mux (first [(cons x_1 xs) (cons y_1 ys)]) (second[(cons x_1 xs) (cons y_1 ys)])) {dmx2}
where [xs ys] = (dmx [x_2 y_2 ... x_{n+1} y_{n+1}]) note: defining xs, ys
= (mux (cons x_1 xs) (cons y_1 ys)) {fst}, {snd}
= (cons x_1 (cons y_1 (mux (first (dmx[x_2 y_2 ... x_{n+1} y_{n+1}])) (second(dmx[x_2 y_2 ... x_{n+1} y_{n+1}])))))) {def'n xs, ys above}
= (cons x_1 (cons y_1 [x_2 y_2 ... x_{n+1} y_{n+1}])) $E(n)$
= [x_1 y_1 x_2 y_2 ... x_{n+1} y_{n+1}] {cons} twice

– Odd base case: $O(0)$.

	(mux (first (dmx $[x_1]$)) (second (dmx $[x_1]$)))	
=	(mux (first $[[x_1]$ nil]) (second $[[x_1]$ nil]))	{dmx1}
=	(mux x_1 nil)	{fst}, {rst}
=	$[x_1]$	{mux0y}

– Odd inductive case: $O(n) \rightarrow O(n+1)$.

	(mux (first (dmx $[x_1\ y_1\ x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$)) (second(dmx $[x_1\ y_1\ x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$)))	
=	(mux (first [(cons x_1 xs) (cons y_1 ys)]) (second[(cons x_1 xs) (cons y_1 ys)]))	{dmx2}
	where $[xs\ ys]$ = (dmx $[x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$)	defining xs, ys
=	(mux (cons x_1 xs) (cons y_1 ys))	{fst}, {snd}
=	(cons x_1 (cons y_1 (mux (first (dmx$[x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$)) (second(dmx$[x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$)))))	{def'n xs, ys}
=	(cons x_1 (cons y_1 $[x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$))	$E(n)$
=	$[x_1\ y_1\ x_2\ y_2\ \ldots x_{n+1}\ y_{n+1}\ x_{(n+1)+1}]$	{cons} twice

8. Theorem: (dmx2 xs) = (dmx xs).

   We divide the proof into two cases:

   – the case in which the operand is the empty list: nil

   – the case when the operand is a non-empty list: $[x_1\ x_2\ \ldots x_{n+1}]$

   The proofs follow.

   – Empty operand case:

	(dmx2 nil)	
=	[nil nil]	{dmx2-0x}
=	(dmx nil)	{dmx0}

   – Non-empty operand case: Proof by induction.

   $D(n) \equiv$ ((dmx2 $[x_1\ x_2\ x_3\ \ldots x_{n+1}]$) = (dmx $[x_1\ x_2\ \ldots x_{n+1}]$))

   * Base case: $D(0)$

	(dmx2 $[x_1]$)	
=	(dmx2 (cons x_1 nil))	{cons}
=	[(cons x_1 xs) ys]	{dmx2-1x}
	where $[ys\ xs]$ = (dmx2 nil)	note: defining xs, ys
=	[nil nil]	{dmx2-0x}
=	[(cons x_1 nil) nil]	{def'n xs, ys}
=	$[[x_1]$ nil]	{cons}
=	(dmx $[x_1]$)	{dmx1}

   * Inductive case: $D(n) \rightarrow D(n+1)$

	(dmx2 [x_1 x_2 x_3 ... $x_{(n+1)+1}$])	
=	(dmx2 (cons x_1 [x_2 x_3 ... $x_{(n+1)+1}$]))	{cons}
=	[(cons x_1 xs) ys]	{dmx2-1x}
	where [ys xs] = (dmx2 [x_2 x_3 ... $x_{(n+1)+1}$])	note: defining xs, ys
	= (dmx [x_2 x_3 ... $x_{(n+1)+1}$])	$D(n)$
	= [(cons x_2 ys) xs]	{dmx2} (or, if $n = 0$, {dmx1})
=	[(cons x_1 xs) (cons x_2 ys)]	def'n ys
=	(dmx [x_1 x_2 x_3 ... $x_{(n+1)+1}$])	{dmx2}

10.1. Insertion Sort

2. Proofs by cases:

   - Case 0: operand is empty

	(len (isort nil))	
=	(len nil)	{isrt0}

   - Case 1: operand is not empty

     Proof by induction:

     $S(n) \equiv$ ((len (isort [x_1 x_2 ... x_{n+1}])) = (len [x_1 x_2 ... x_{n+1}]))

     * Base case: $S(0)$

	(len (isort [x_1]))	
=	(len (isort (cons x_1 nil)))	{cons}
=	(len (cons x_1 nil))	{isrt1}
=	(len [x_1])	{cons}

     * Inductive case: $S(n) \rightarrow S(n+1)$

	(len (isort [x_1 x_2 x_3 ... $x_{(n+1)+1}$]))	
=	(len (isort (cons x_1 (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))	{cons} twice
=	(len (insert x_1 (isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))	{isrt2}
	Case 1: if $x_1 \leq$ (first(isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))	
=	(len (cons x_1 (isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))	{ins1}
=	1 + (len (isort (cons x_2 [x_3 ... $x_{(n+1)+1}$])))	{len1}
=	$1 + (n + 1)$	$S(n)$
=	$(n + 1) + 1$	algebra
	Case 2: if $x_1 >$ (first(isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))	
=	(len (cons (first(isort (cons x_2 [x_3 ... $x_{(n+1)+1}$])))	{ins2}
	(insert x_1 (rest (isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))))	
=	1 + (len (insert x_1 (rest (isort (cons x_2 [x_3 ... $x_{(n+1)+1}$]))))))	{len1}
=	$1 + (1 +$ (len (rest (isort [x_2 x_3 ... $x_{(n+1)+1}$]))))	{ins-len}
=	$1 + (1 + ((n + 1) - 1))$	$S(n)$
	(also using: (len (rest xs)) = (len xs) - 1 if (consp xs))	
=	$(n + 1) + 1$	algebra

4. a) zero (because no value can occur in the list if it has no elements).

   b)
```
(defthm ct-with-new-x
        (= (ct (cons x xs))
           (+ (ct x xs) 1)))
```

c)
```
(defthm ct-with-new-y-not-x
        (implies (not (equal y x))
                 (= (ct x (cons y xs))
                    (ct x xs))))
```

d)

(ct x nil)	$= 0$	{ct0}
(ct x (cons x xs))	$= 1 +$ (ct x xs)	
(ct x (cons y xs))	$=$ (ct x xs)	if $y \neq x$

The operator ct can be defined in ACL2 as follows:

```
(defun ct (x xs)
  (if (not (consp xs))
      0
      (if (equal x (first xs))
          (+ (ct x (rest xs)) 1)
          (ct x (rest xs)))))
```

6. The theorem is stated as follows:

```
(defthm isort-delivers-permutation
   (permp (isort xs) xs))
```

Here is a complete proof script in ACL2.

Note: Code for msort that ACL2 will admit to its logic is included in the script, even though it does not relate to the *isort* exercise. This is facilitate experimenting with msort computationally. The msort code provided here refers to a version of the demultiplex operator, dmx, derived from the equations for an alternative demultiplex operator, dmx2, defined in Box 9.3, p.175. ACL2 runs into some problems admitting msort when it refers to the demultiplex operator defined on p.173. When it is practicable, the book will address this problem. Until then, it would be advisable to use the version of dmx defined here to carry out any msort computations in ACL2.

```
(in-package "ACL2")
(defun dmx (xys)                ; xys = (x1 y1 x2 y2 x3 y3 ...)
  (if (consp xys)
      (let* ((x1 (first xys))
             (yxs (rest xys))   ; yxs  = (y1 x2 y2 x3 y3 x4 ...)
             (ysxs (dmx yxs))   ; ysxs = ((y1 y2 ...)(x2 x3 ...))
             (ys  (first ysxs)) ; ys   = (y1 y2 ...)
             (xs (second ysxs)); xs   = (x2 x3 ...)
        (list (cons x1 xs) ys)) ; {dmx1} ((x1 x2 ...)(y1 y2 ...))
      (list xys xys)))          ; {dmx0}
(defun mrg (xs ys)
  (declare (xargs :measure (+ (len xs) (len ys))))
  (if (and (consp xs) (consp ys))
```

```
      (let* ((x (first xs)) (y (first ys)))
        (if (<= x y)
            (cons x (mrg (rest xs) ys))     ; {mg<}
            (cons y (mrg xs (rest ys)))))   ; {mg>}
      (if (consp xs)
          xs         ; note: ys is empty    ; {mg1}
          ys)))      ; note: xs is empty    ; {mg0}
(defthm dmx-shortens-list ; lemma to help ACL2 admit def'n of msort
  (implies (consp (rest xs))
           (let* ((odds-evens (dmx xs)) ; xs = (x1 x2 ...)
                  (odds       (first odds-evens))
                  (evns       (second odds-evens)))
             (and (< (len odds) (len xs))
                  (< (len evns) (len xs))))))
(defun msort (xs)
  (declare (xargs
            :measure (len xs)
            :hints (("Goal"
                     :use ((:instance dmx-shortens-list))))))
  (if (consp (rest xs)) ; (len xs) > 1?
      (let* ((odds-evens (dmx xs)) ; xs = (x1 x2 ...)
             (odds       (first odds-evens))
             (evns       (second odds-evens)))
        (mrg (msort odds) (msort evns))) ;    {m-s2}
      xs))               ; xs = (x1) or empty {m-s1}

(defun insert (x xs) ; assume xs is already in proper order
  (if (and (consp xs) (> x (first xs)))
      (cons (first xs) (insert x (rest xs))) ; {ins2}
      (cons x xs)))                          ; {ins1}
(defun isort (xs)
  (if (consp (rest xs)) ; (len xs) > 1?
      (insert (first xs) (isort (rest xs)))  ; {isrt2}
      xs))                                   ; {isrt1}

(defun up (xs)
   (or (not (consp (rest xs)))
       (and (<= (first xs) (second xs))
            (up (rest xs)))))
(defthm mrg-preserves-order
  (implies (and (up xs) (up ys))
           (up (mrg xs ys))))
(defthm msort-sorts-base-case
  (implies (not(consp(rest xs)))
           (up (msort xs))))
(defthm msort-sorts
  (up (msort xs)))
(defthm dmx-preserves-len
```

```
  (let* ((xsys (dmx xys))
         (odds (first xsys))
         (evns (second xsys)))
    (= (+ (len odds) (len evns))
       (len xys))))
(defthm mrg-preserves-len
  (= (len (mrg xs ys)) (+ (len xs) (len ys))))
(defthm msort-preserves-len-base-case
  (implies (not(consp(rest xs)))
           (= (len (msort xs))
              (len xs))))
(defthm msort-preserves-len>=1-case
   (= (len (msort(cons x xs)))
     (1+ (len (msort xs)))))
(defthm msort-preserves-len
  (= (len (msort xs)) (len xs)))
(defthm mrg-conservation-of-values
  (iff (member-equal e (mrg xs ys))
       (or (member-equal e xs) (member-equal e ys))))
;(defthm msort-conservation-of-values
;  (iff (member-equal e xs)
;       (member-equal e (msort xs))))

(defun del (x xs)
   (if (not(consp xs))
       nil
       (if (equal x (first xs))
           (rest xs)
           (cons (first xs) (del x (rest xs))))))
(defun occurs-in (x xs)
   (if (consp xs)
       (or (equal x (first xs))
           (occurs-in x (rest xs)))
       nil))
(defun permp (xs ys)
  (if (not(consp xs))
      (not (consp ys))
      (and (occurs-in (first xs) ys)
           (permp (rest xs) (del (first xs) ys)))))

; ex 6, sec 10.1, p.181
(defthm isort-delivers-permutation
   (permp (isort xs) xs))
```

10.2. Order-Preserving Merge

1. (defthm mrg-len-thm

```
(= (len (mrg xs ys))
   (+ (len xs) (len ys))))
```

2. Proof of mrg-len-thm by induction.

   $L(n) \equiv \forall j.((\mathsf{len}\ (\mathsf{mrg}\ [x_1\ x_2\ \ldots x_j]\ [y_{j+1}\ y_{j+2}\ \ldots y_n]) = n)$.

   Note: Domain of discourse of $\forall j$ is 0, 1, ..., n.

   - Base case: $L(0)$.

	(len (mrg nil nil))	
=	(len nil)	{mg0}
=	0	{len0}
=	0+0	algebra
=	(len nil) + (len nil)	{len0} twice

   - Inductive case: $L(n) \rightarrow L(n+1)$.

	(len (mrg $[x_1\ x_2\ \ldots x_j]$ $[y_{j+1}\ y_{j+2}\ \ldots y_{n+1}]$))	
	Case 1: if $j = 0$	
=	(len $[y_1\ y_2\ \ldots y_{n+1}]$)	{mg0}
=	$n+1$	
	Case 2: if $j = n+1$	
=	(len $[x_1\ x_2\ \ldots x_{n+1}]$)	{mg1}
=	$n+1$	
	Case 3: if $0 < j < n+1$	
=	(len (mrg $[x_1\ x_2\ \ldots x_j]$ $[y_{j+1}\ y_{j+2}\ \ldots y_{n+1}]$))	{mg1}
	Case 3a: if $0 < j < n+1$ and $x_1 \leq y_{j+1}$	
=	(len (cons x_1 (mrg $[x_2\ \ldots x_j]$ $[y_{j+1}\ y_{j+2}\ \ldots y_{n+1}]$)))	{mg1}
=	1 + (len (mrg $[x_2\ \ldots x_j]$ $[y_{j+1}\ y_{j+2}\ \ldots y_{n+1}]$))	{len1}
=	$1+n$	$L(n)$
=	$n+1$	algebra
	Case 3b: if $0 < j < n+1$ and $x_1 > y_{j+1}$	
=	(len (cons y_{j+1} (mrg $[x_1\ x_2\ \ldots x_j]$ $[y_{j+2}\ \ldots y_{n+1}]$)))	{mg2}
=	1 + (len (mrg $[x_1\ x_2\ \ldots x_j]$ $[y_{j+2}\ \ldots y_{n+1}]$))	{len1}
=	$1+n$	$L(n)$
=	$n+1$	algebra

4.
```
(defthm mrg-val-thm
  (iff (or (occurs-in z xs) (occurs-in z ys))
       (occurs-in z (mrg xs ys)))
```

11.6. Inserting a New Item in a Search Tree

2. ---

   (ins 1125 "Modem"
   [8444 "Audio Card" nil nil])
   ⇓
   [8444 "Audio Card" [1125 "Modem" nil nil] nil]

   (ins 4878 "Mouse"

```
    [8444 "Audio Card" [1125 "Modem" nil nil] nil])
    ⇓
[4878 "Mouse" [1125 "Modem" nil nil]
              [8444 "Audio Card" nil nil]]
```

```
(ins 2088 "Laser Jet"
    [4878 "Mouse" [1125 "Modem" nil nil]
                  [8444 "Audio Card" nil nil]])
    ⇓
[4878 "Mouse" [2088 "Laser Jet" [1125 "Modem" nil nil] nil]
              [8444 "Audio Card" nil nil]]
```

4. $H((\text{height s})) \equiv (\text{height } (\text{hook } x\ a\ s)) \leq (\text{height } s) + 1$

   Proof by strong induction on $(\text{height } s)$

Base case: $(\text{height } s) = 0$, Note: $s = \text{nil}$ because nil is the only tree of height zero

	$(\text{height } (\text{hook } x\ a\ s))$	
$=$	$(\text{height } (\text{hook } x\ a\ \text{nil}))$	$s = \text{nil}$
$=$	$(\text{height } (\text{mktr } x\ a\ \text{nil nil}))$	{hook0}
$=$	$(\text{height } [x\ a\ \text{nil nil}])$	{defun mktr}
$=$	$1 + (\text{max } (\text{height nil}) (\text{height nil}))$	{ht1}
$=$	$1 + (\text{max } 0\ 0)$	{ht0} twice
$=$	$1 + 0$	{algebra}
$=$	$1 + (\text{height nil})$	{ht0}
$=$	$(\text{height } s) + 1$	$s = \text{nil}$, {algebra}

Inductive case: $(\text{height } s) = n + 1$
Since $(\text{height } s) \geq 1$, assume $s = [k\ d\ lf\ rt]$. Note: $(\text{height } lf) < (\text{height } s)$, $(\text{height } rt) < (\text{height } s)$

	$(\text{height } (\text{hook } x\ a\ s))$		
$=$	$(\text{height } (\text{hook } x\ a\ [k\ d\ lf\ rt]))$		$s = [k\ d\ lf\ rt]$
$=$	$(\text{height } (\text{mktr } k\ d\ (\text{hook } x\ a\ lf)\ rt))$	if $x < k$	{hook¡}
	$(\text{height } (\text{mktr } k\ d\ lf\ (\text{hook } x\ a\ rt)))$	if $x > k$	{hook¿}
	$(\text{height } (\text{mktr } x\ a\ lf\ rt))$	if $x = k$	{hook=}
$=$	$(\text{height } [k\ d\ (\text{hook } x\ a\ lf)\ rt])$	if $x < k$	{defun mktr}
	$(\text{height } [k\ d\ lf\ (\text{hook } x\ a\ rt)])$	if $x > k$	{defun mktr}
	$(\text{height } [x\ a\ lf\ rt])$	if $x = k$	{defun mktr}
$=$	$1 + (\text{max } (\text{height } (\text{hook } x\ a\ lf)) (\text{height } rt))$	if $x < k$	{ht1}
	$1 + (\text{max } (\text{height } lf) (\text{height } (\text{hook } x\ a\ rt)))$	if $x > k$	{ht1}
	$1 + (\text{max } (\text{height } lf) (\text{height } rt))$	if $x = k$	{ht1}
$\leq$	$1 + (\text{max } ((\text{height } lf) + 1) (\text{height } rt))$	if $x < k$	$\{H((\text{height } lf))\}$
	$1 + (\text{max } (\text{height } lf) ((\text{height } rt) + 1))$	if $x > k$	$\{H((\text{height } rt))\}$
	$1 + (\text{max } (\text{height } lf) (\text{height } rt))$	if $x = k$	{same formula}
$\leq$	$1 + (1 + (\text{max } (\text{height } lf) (\text{height } rt)))$	if $x < k$	{algebra}
	$1 + (1 + (\text{max } (\text{height } lf) (\text{height } rt)))$	if $x > k$	{algebra}
	$1 + (1 + (\text{max } (\text{height } lf) (\text{height } rt)))$	if $x = k$	{algebra}
$\leq$	$1 + (\text{height } [k\ d\ lf\ rt])$		{ht1}, {defun lft}, {defun rgt}
$=$	$(\text{height s}) + 1$		$\{s = [k\ d\ lf\ rt]\}$, {algebra}

11.8 Double Rotations

2.

$$
\begin{array}{lll}
 & (\mathsf{ordp}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]) & \\
= & (\mathsf{emptyp}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]) \vee & \{\text{defun ordp}\} \\
 & ((\mathsf{treep}\ [\mathsf{k}\ \mathsf{d}\ \mathsf{nil}\ \mathsf{nil}]) \wedge & \\
 & (\forall x.((\mathsf{keyp}\ \mathsf{x}\ \mathsf{nil}) \rightarrow x < (\mathsf{key}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]))) \wedge (\mathsf{ordp}\ (\mathsf{lft}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}])) \wedge & \\
 & (\forall y.((\mathsf{keyp}\ \mathsf{y}\ \mathsf{nil}) \rightarrow y > (\mathsf{key}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]))) \wedge (\mathsf{ordp}\ (\mathsf{rgt}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]))) & \\
= & \mathit{false} \vee & \{\text{defun emptyp}\} \\
 & (\mathit{true} \wedge & \{\text{defun treep}\} \\
 & (\forall x.(\mathit{false} \rightarrow x < (\mathsf{key}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]))) \wedge (\mathsf{ordp}\ \mathsf{nil}) \wedge & \{\text{defun keyp}\}, \{\text{defun lft}\} \\
 & (\forall y.(\mathit{false} \rightarrow y > (\mathsf{key}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]))) \wedge (\mathsf{ordp}\ \mathsf{nil})) & \{\text{defun keyp}\}, \{\text{defun rgt}\} \\
= & \mathit{false} \vee & \\
 & (\mathit{true} \wedge & \\
 & (\forall x.\mathit{true}) \wedge \mathit{true} \wedge & \{\rightarrow \text{truth tbl}\}, \{\text{defun ordp}\} \\
 & (\forall y.\mathit{true}) \wedge \mathit{true}) & \{\rightarrow \text{truth tbl}\}, \{\text{defun ordp}\} \\
= & \mathit{false} \vee & \\
 & (\mathit{true} \wedge \mathit{true} \wedge & \{\text{def'n } \forall\} \\
 & \mathit{true} \wedge \mathit{true}) & \{\text{def'n } \forall\} \\
= & \mathit{false} \vee \mathit{true} & \{\wedge \text{ truth table}\} \\
= & \mathit{true} & \{\vee \text{ truth table}\}
\end{array}
$$

$$
\begin{array}{lll}
 & (\mathsf{balp}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]) & \\
= & (\mathsf{emptyp}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}]) \vee & \{\text{defun balp}\} \\
 & (((\mathsf{abs}\ ((\mathsf{height}\ (\mathsf{lft}[k\ d\ \mathsf{nil}\ \mathsf{nil}])) - (\mathsf{height}\ (\mathsf{rgt}[k\ d\ \mathsf{nil}\ \mathsf{nil}])))) \leq 1) \wedge & \\
 & (\mathsf{balp}\ (\mathsf{lft}\ [k\ d\ \mathsf{nil}\ \mathsf{nil}])) \wedge (\mathsf{balp}\ (\mathsf{rgt}[k\ d\ \mathsf{nil}\ \mathsf{nil}]))) & \\
= & \mathit{false} \vee & \{\text{defun emptyp}\} \\
 & (((\mathsf{abs}\ ((\mathsf{height}\ \mathsf{nil}) - (\mathsf{height}\ \mathsf{nil}))) \leq 1) \wedge & \{\text{defun lft}\}, \{\text{defun rgt}\} \\
 & (\mathsf{balp}\ \mathsf{nil}) \wedge (\mathsf{balp}\ \mathsf{nil})) & \{\text{defun lft}\}, \{\text{defun rgt}\} \\
= & \mathit{false} \vee & \\
 & (((\mathsf{abs}\ (0 - 0)) \leq 1) \wedge & \{\text{defun height}\} \text{ twice} \\
 & \mathit{true} \wedge \mathit{true}) & \{\text{defun balp}\} \text{ twice} \\
= & \mathit{false} \vee & \\
 & (\mathit{true} \wedge & \{\text{algebra}\} \\
 & \mathit{true} \wedge \mathit{true}) & \\
= & \mathit{false} \vee \mathit{true} & \{\wedge \text{ truth table}\} \\
= & \mathit{true} & \{\vee \text{ truth table}\}
\end{array}
$$

4.
```
(defun rot- (s)    ; rotate counterclockwise if too tall on right
  (let* ((k (key s)) (d (dat s))    ; rot- assumes s is not empty
         (kL (lft s)) (kR (rgt s)))
    (if (> (height kR) (+ (height kL) 1))           ; unbalanced?
        (if (> (height (lft kR)) (height (rgt kR))) ; inside rgt?
            (zig(mktr k d kL (zag kR)))             ; dbl rotate
            (zig s))                                ; sngl rotate
        s)))                                        ; no rotate
```