CS313K Notes

J Strother Moore
The University of Texas at Austin

November 1, 2024

Copyright ©2024 by J Strother Moore

All rights reserved. I hereby grant permission for these notes to be printed, photo-copied, and/or stored electronically for personal or classroom use.

To Jo

Contents

Preface **xi**

0.1 About these Notes (Assignment 1: 5 days) xi

0.2 About this Course (Assignment 1 cont'd) xiii

1 A Little Greek (Assignment 1 cont'd) 1

2 The ACL2 Programming Language (Assignment 1 cont'd) 3

2.1 Evaluating Simple Expressions (Assignment 1 cont'd) 6

2.2 Defining Functions (Assignment 1 cont'd) 9

2.3 True, False, Apples, and Firetrucks (Assignment 2: 2 days) 12

2.4 Evaluating Calls of Defined Functions (Assignment 2 cont'd) 13

2.5 Recursion (Assignment 2 cont'd) 16

2.6 Practice with Definitions (Assignment 2 cont'd) 18

2.7 Lessons from the Practice (Assignment 3: 5 days) 28

2.8 About Recursion (Assignment 3 cont'd) 29

2.9 Comments and Displaying Code (Assignment 3 cont'd) 30

2.10 The Five Types of Objects (Assignment 3 cont'd) 34

2.11 Primitive Functions (Assignment 4: 2 days) 48

2.12 Some Simple Definitions (Assignment 4 cont'd) 49

2.13 Defining New Functions (Assignment 4 cont'd) 57

2.14 Recursion on Lists (Assignment 5: 5 days) 59

2.15 Practice, Practice, Practice (Assignment 6: 7 days) 73

2.16 Abbreviations (Assignment 6 cont'd) 80

3 The Need for Logic (Assignment 7: 2 days) 83

3.1 An Informal Proof (Assignment 7 cont'd) 83

3.2 Implication (Assignment 7 cont'd) 92

3.3 Proving Another Simple Property (Assignment 7 cont'd) 93

3.4 Replacement of "Equals" by "Equals" (Assignment 8: 5 days) 96

3.5 What's Going On Here? (Assignment 8 cont'd) 100

3.6 Recognizing and Using Truths (Assignment 8 cont'd) 101

3.7 Useful Identities (Assignment 8 cont'd) 105

3.8 Summary (Assignment 8 cont'd) 108

4 Logic (Assignment 9: 7 days) 109

4.1 Preliminaries (Assignment 9 cont'd) 109

4.2 Syntax (Assignment 9 cont'd) 117

4.3 Axioms (Assignment 10: 2 days) 125

4.4 Rules of Inference (Assignment 10 cont'd) 127

4.5 Tautologies (Assignment 10 cont'd) 131

4.6 About the Rules of Inference (Assignment 11: 7 days) 134

4.7 Writing Proofs Down (Assignment 11 cont'd) 143

4.8 Proofs of Tautologies (Assignment 11 cont'd) 150

4.9 Counterexamples, Witnesses, and Models (Assignment 12: 14 days) 157

4.10 Practice Proofs (Assignment 12 cont'd) 158

5 Structural Induction (Assignment 13: 12 days) 169

5.1 New Rule of Inference (Assignment 13 cont'd) 170

5.2 Advice About Induction (Assignment 13 cont'd) 171

5.3 Practice Proofs (Assignment 13 cont'd) 176

6 The Elementary Arithmetic Waiver (Assignment 14: 7 days) 189

7 Quantifiers (Assignment 14 cont'd) 191

7.1 The Universal Quantifier (Assignment 14 cont'd) 191

7.2 The Existential Quantifier (Assignment 14 cont'd) 196

7.3 ASCII Substitutes for Quantifiers (Assignment 14 cont'd) 198

7.4 Talking About Quantifiers Precisely (Assignment 15: 2 days) 199
7.5 Imprecise Talk about the Rules of Inference (Assignment 15 cont'd) 204
7.6 Rules of Inference for Quantified Formulas (Assignment 16: 7 days) 210
7.7 Practice Proofs (Assignment 16 cont'd) 213

8 Set Theory (Assignment 17: 5 days) 243
8.1 Basic Set Notation (Assignment 17 cont'd) 245
8.2 Set Operations (Assignment 17 cont'd) 248
8.3 Extensions to Quantifier and Set Builder Notation (Assignment 18: 2 days) 254
8.4 Pair Notation (Assignment 18 cont'd) 256
8.5 Generalized Union and Intersection (Assignment 18 cont'd) 257
8.6 Cross Product (Assignment 18 cont'd) 259
8.7 Relations (Assignment 19: 7 days) 260
8.8 Orders (Assignment 19 cont'd) 264
8.9 Equivalence Relations (Assignment 19 cont'd) 265
8.10 Uniqueness Notation (Assignment 20: 5 days) 267
8.11 Partitions (Assignment 20 cont'd) 267
8.12 Equivalence Relations v Partitions (Assignment 20 cont'd) 269
8.13 Functions (Assignment 20 cont'd) 270
8.14 Cardinality (Assignment 20 cont'd) 274

Appendices 274

A A Potted History of Logic in CS 275

B On Congruences 279

C On Entailment 283

D ASCII Substitutes 285

Bibliography 287

Index 289

Preface

0.1 About these Notes (Assignment 1: 5 days)

These notes were written when I was teaching CS313K *Logic, Sets, and Functions*, a required undergraduate course in the Computer Science Department of the University of Texas at Austin. But, as mentioned in Appendix A, mathematical logic has long been thought of as "pure math" as opposed to "applied math." The majority of logic textbooks explore the power and limits of logic. But I don't do that here. Instead, I'm more interested in using logic to describe the intended operation of digital devices and the algorithms running on them. So rather than teach CS313K out of a traditional logic textbook, I wrote these notes to focus mainly on those parts of logic I think are useful to computer science practitioners.

About 150 students were enrolled for each course offering. The class met twice a week. Each week I assigned reading and homework problems. I would open each lecture with the question "Does anyone have any questions about the assigned reading?" After all their questions had been answered, I gave a quiz. The quiz was always multiple choice and students would indicate their answers with a remote control that would register and record each student's answer. Students were allowed to work together, partly because it was impossible to prevent them from doing so but also because it was often good for students to justify their answers to another student. I'd then display the distribution of answers and if a sizable number of students got the answer wrong, I'd give a mini-lecture on the subject. Then we'd move on to the next question. These quizzes counted for a significant fraction of each student's grade. This had the good effect of basically requiring students to come to class, to have read the relevant material in advance, and to clarify their understanding by asking me questions before the quiz.

After I retired from teaching I realized that the notes might be valuable to anybody with a computer science background who might want to learn a little practical logic. But because of the way they were used in class, a few remarks are helpful.

Each chapter and major section includes an "assignment" number. For example, you'll note that the section heading above reads "About these Notes (Assignment 1: 5 days)" and the next section heading below reads "About this Course (Assignment 1 cont'd)." When the course was being taught the students got these notes on the first Thursday of the semester and were told to read assignment 1 before the next class meeting, which was the following Tuesday – five days hence. Assignment 1 covered the Preface, Chapter 1, and the first few

sections of Chapter 2, so you'll see "Assignment 1 cont'd" in each section heading down to Section 2.3. Since I believe that you learn a skill by practicing it, so I told students to answer as many of the questions in the assignment as they could. This prepared them for the quiz in the next class meeting. Some questions – the ones in uppercase – were assigned to be turned in before the next class meeting.

Because we only met on Tuesdays and Thursdays the time allotted varies but frequently you'll notice the pattern is 5 days, 2 days, 5 days, 2 days, etc. Occassionally you'll see 7 days or even 14, because the material takes longer to cover or holidays interrupted our classes.

In any case, I wouldn't be too concerned with these assignment numbers, but I kept them in the notes because they might help some readers move along through the material. You should work at your own pace. And, by the way, you're free to use previously-posed theorems in your proofs, even if you didn't figure out their proofs.

The other thing I should note is that you should install ACL2 on your laptop. See the ACL2 Homepage for installation instructions. I run ACL2 in an Emacs shell buffer. Some students preferred the Eclipse interface, called the "ACL2 Sedan" or ACL2s. See ACL2s for installation instructions. Proofs in this course are just old-fashioned "hand proofs." We won't use ACL2 to prove theorems. But we will use it to evaluate expressions and as a language in which we'll define functions to prove things about.

Since ACL2 is Lisp, you'll note that I write such things as `(f x (g y))` where more commonly you'd see $f(x, g(y))$. One reason I do that is to allow you to use ACL2 to evaluate expressions to learn their meanings by example. Then, using ACL2 syntax, I lay out the logic we'll use to do proofs: axioms and rules of inference. After a bit of practice at proofs in that system, I loosen the rules and allow you to use infix notation for certain very familiar operations, like allowing `x + (2 * y)` for `(+ x (* 2 y))`. This gives rise to an unusual mix of infix and Lisp notation that makes no sense to the ACL2 system. But over the years I have found that when I write proofs out on paper I often abbreviate ACL2 terms with infix. Finally, some students preferred to type their answers to homework problems and that raised the question of how they should type certain special symbols, like $\leq$ and $\rightarrow$. Because homeworks were graded by teaching assistants in the course offerings of CS313K, it was important to get uniformity on these matters. So the notes recommend some ASCII substitutes for many special symbols used in the notes. I left those recommendations in the notes, in case you prefer to type answers to the homeworks.

I apologize for not providing an answer key for the homeworks. Perhaps someday.

So be aware that nobody but you will grade your homeworks! You're on your own. Handle homeworks how ever you want. But my recommendation is that you always write as clearly to yourself as you would to somebody grading you! It is easy to imagine meaning in random scribbles – meaning that is not there or is at best ambiguous. When you take care to write down exactly what you mean you have a much better chance of recognizing mistakes and correcting them. Expressing yourself clearly is a skill that will serve you well and the only way to develop that skill is to practice it.

0.2 About this Course (Assignment 1 cont'd)

This course is not about programming. It is about symbolic logic [4, 3, 2]. Symbolic logic is the predominant mathematical system in computer science. As with other sciences and engineering disciplines, we use mathematics to model and analyze complex systems. Unlike those other disciplines, where the "right" mathematics is calculus or statistics, in computer science the "right" mathematics is mathematical logic. It is perfect for modeling and analyzing digital systems.

To understand physics you must understand calculus. To understand computer science you must understand mathematical logic.

For a potted history of logic and its role in computer science, see Appendix A.

But mathematical logic was designed by logicians to be studied by logicians, not used as a practical mathematical system! In computer science, logic is used to establish truth: we prove things with it.

I didn't want to teach logic from a standard textbook. I decided to write these notes so that I could stress how computer scientists use logic. My logician friends will find my treatment of this glorious subject superficial. I deal with none of the deep philosophical questions answered by the study of the logic. My rules of inference are incredibly elaborate and complicated compared to the ones you'll see in a typical introduction to logic. That's because they're designed to be used, not studied. I have one goal: teach you how to prove things about computational systems and I intend for you to be able to use the rules I give you.

I start by teaching you a really simple programming language, called ACL2. *But it is important that you get comfortable running ACL2.* All we'll be doing with ACL2 is defining and running really simple programs, like programs to search a list for an element, count the number of times an element occurs, etc. While the ACL2 system supports much more than just defining and running programs, that's all you have to learn. But if you don't learn how to write simple ACL2 programs, you probably won't pass the course!

Why am I making you learn another programming language? To learn to use logic to talk about computer science applications we need some simple applications. You're going to write those simple applications in ACL2 because programs in it are much simpler to analyze than those of other programming languages you might know. Here's an analogy. When you learn to use calculus to model the physical world, you start by using it to model objects falling in a vacuum, not helicopters in hurricanes. I should add that no computer science student should be worried about learning another programming language! You will learn a dozen before you graduate and you will continue to learn new ones throughout your careers. Get used to it!

Once you've learned how to write simple applications in ACL2, we'll start talking and reasoning about them with logic.

Acknowledgements: I would like to acknowledge the help I've received from Bob Boyer, Matt Kaufmann, Pete Manolios, and Sandip Ray for developing parts of this book. In

addition, I'm grateful to the graduate teaching assistants from the Spring 2009 offering of CS 313K, namely David Rager, Behnam Robatmili, Ian Wehrman, and Nathan Wetzler, for many helpful comments. I also appreciate the efforts of my undergraduate assistants, including David Garcia, Sarah Imboden, Jarmel James, Ryan King, Young-Suk Lee, Lon Nix, and Alian Rodriguez. I owe a debt of gratitude to the many students who have read various editions of this book. Most of all, I thank my wife, Jo, for putting up with me while I wrote and repeatedly revised this text.

1

A Little Greek (Assignment 1 cont'd)

You will see a lot of Greek symbols in computer science. You should know their names.

name	*symbol*	*alternative symbol*	*uppercase version*	*name*	*symbol*	*alternative symbol*	*uppercase version*
alpha	α		A	nu	ν		N
beta	β		B	xi	ξ		Ξ
gamma	γ		Γ	omicron	o		O
delta	δ		Δ	pi	π	ϖ	Π
epsilon	ϵ	ε	E	rho	ρ	ϱ	P
zeta	ζ		Z	sigma	σ	ς	Σ
eta	η		H	tau	τ		T
theta	θ	ϑ	Θ	upsilon	υ		Υ
iota	ι		I	phi	ϕ	φ	Φ
kappa	κ		K	chi	χ		X
lambda	λ		Λ	psi	ψ		Ψ
mu	μ		M	omega	ω		Ω

• **Question 1**: What is the name of this symbol: ϕ? •

• **Question 2**: What is the name of this symbol: ψ? •

• **Question 3**: What is the name of this symbol: γ? •

• **Question 4**: What is the name of this symbol: α? •

• **Question 5**: What is the name of this symbol: ω? •

• **Question 6**: What is the name of this symbol: δ? •

• **Question 7**: What is the name of this symbol: π? •

• **Question 8**: What is the name of this symbol: σ? •

• **Question 9**: What is the name of this symbol: β? •

Why do we use Greek letters? In computer science, we spend a lot of time talking about about languages, e.g., Java, HTML, C++, etc. This can get confusing because we use one language (usually English) to talk about another language, e.g., Java. If we use English to talk about Java, we say Java is the *object language* and English is the *metalanguage*. (Think

of "meta" meaning "about," e.g., meta-data is data about the data in a file.)

So for example, a computer scientist might say "A 'simple conditional assignment' is a Java statement of the form '`if (`ϕ`) {`α` = `β`;};`' where α is a Java variable."

According to this definition, the Java statements '`if (k<0) {k=k+1;};`' and '`if (x.name == y.name) {temp=x.age;};`' are both simple conditional assignments because they match the description of the pattern shown in the English definition. In the first, ϕ is understood to be the Java snippet '`k<0`' and in the second ϕ is '`x.name == y.name`'.

• **Question 10**: In the simple conditional assignment statement '`if (k<0) {k=k+1;};`', what is α? •

• **Question 11**: In the simple conditional assignment statement '`if (k<0) {k=k+1;};`', what is β? •

• **Question 12**: In the simple conditional assignment statement '`if (x.name == y.name) {temp=x.age;};`', what is α? •

• **Question 13**: In the simple conditional assignment statement '`if (x.name == y.name) {temp=x.age;};`', what is β? •

Notice how we're using English to talk about Java. The Greek letters in the definition of a 'simple conditional assignment' are called *metavariables* because they are variables in English (the metalanguage). Their values are snippets of Java (the object language).

So why do we use Greek letters for metavariables? Just imagine how confusing it would be if I had said "A 'simple conditional assignment' is a Java statement of the form '`if (test) {v = expr;};`' where `v` is a Java variable."

Do you see the problem? Now you can't tell what part of my definition is meant to be Java and what part is meant to be English. For example, does a simple conditional assignment statement have to test the variable `test` or can it test any Java expression? Must it assign to the Java variable `v` or can it assign to any Java variable? We can't tell from my new definition. Depending on my meaning, '`if (k<0) {k=k+1;};`' may or may not be a simple conditional assignment. Since we tend not to use Greek in our Java programs, using Greek letters for metavariables helps clarify what we mean.

Sometimes we use *italized* variables for the metavariables. For example, we might write the pattern as '`if (`*test*`) {`*v*`=`*expr*`};`.'

2

The ACL2 Programming Language (Assignment 1 cont'd)

ACL2 is a simple programming language. It is a variant of one of the oldest and most important programming languages in the history of computing, Lisp. Created in 1958 to provide a mathematical notation for programs, many innovations in computer science were first explored in Lisp, including the notion of automatic storage allocation and garbage collection, tree-structured data, and Object orientation. Lisp is often used in Artificial Intelligence research and is extremely convenient for rapid prototyping. It has even flown in space on Deep Space 1.

ACL2 is in fact an extension of a small subset of the ANSI standard programming language named Common Lisp. But my goal is not to teach you Lisp or even ACL2. My goal is to teach you how to *think* about computational concepts and how to reason formally about them. With just a tiny subset of ACL2 we can pose some interesting computational problems and then think about them. So I'm going to start by teaching you a tiny subset of ACL2, just enough to let us pose some questions.

Like most programming languages, ACL2 has various kinds of data objects. Numbers, like `-1` and `23`, are examples of ACL2 data objects. The two Boolean objects representing truth and falsity are also data objects. In the first part of this introduction to ACL2, I will limit myself to numbers and Booleans. That will make our examples sort of boring but familiar.

Aside: In these notes I use the word "object" to describe any data value. When I use the capitalized version of the word, "Object," I'm using it in the same sense that Java does, i.e., an Object is an instance of a class, with fields and inherited methods. ACL2 does not have Objects. It only has objects. In the sense that I'm using the words, all Java Objects are objects – they can be passed around and manipulated as data. Java also has objects that are not Objects, namely its so-called "primitive values," the Booleans and the numbers. Another way to think of the distinction is that Objects are mutable (they can be changed) and objects are Platonic, eternal, constant, mathematical entities. No operation will change a mathematical object. I almost never use "Object" because ACL2 doesn't have any. Just remember, an "object" is any data value and ACL2 has several kinds of objects, including Booleans, numbers, and others we'll discuss later. *End of Aside*

Every ACL2 program is a mathematical function: it takes some inputs and produces an output that depends only on the inputs. It produces the same output every time it is called with given inputs. The output is determined by a simple expression that applies other functions to the input in exactly the way you'd compute the value of $sq(5)$ from the

definition

$$sq(n) = n \times n$$

In particular, $sq(5) = (5 \times 5) = 25$.

In one sense, ACL2 is simple to learn because *all a program can do is evaluate (calculate the value of) an expression*! There are no assignments, no classes or methods or fields or Objects as you probably understand them, no while statements, no throws, no catches, no exceptions, no threads, etc. Expressions are built out of variables, constants, and functions calls.

Once you know the constants and the primitive functions, you know everything.[1]

You're probably asking yourself "what can I do with such a dumb language?" You'll probably be surprised.

Learning how to program in such a language will help you program in any language. But our real goal, recall, is learning how to think and reason.

While learning ACL2 it is helpful to write and run sample programs. We'll deal explicitly with expressions and function definitions shortly. But it's important that you have ACL2 running and you're able to experiment with simple definitions and evaluations.

ACL2 presents itself to the user as a read-eval-print loop: it reads an expression, evaluates it, prints the result, and repeats. (ACL2 won't print the values of some special expressions.) Before it reads the next expression it prints a prompt. The prompt typically has one of three forms. It might be "`ACL2 !>`" or "`ACL2 p!>`" or "`ACL2 p>`".

The "`ACL2`" part of the prompt is the "symbol package" you're in. Symbol packages allow users to avoid name clashes. But in this course we'll always stay in the `ACL2` package.

If there is a "`p`" after the symbol package, it means ACL2 is in "program" mode. Otherwise it is "logic" mode. In program mode the system will not try to prove anything. The only time this makes a difference in this course is when you're defining functions. In logic mode, ACL2 will try to prove your recursive definitions always terminate. In program mode, it ignores that requirement.

Finally, if there is an exclamation mark "`!`" in the prompt then the system will check certain type agreements in the expression. For example, when types are being checked, arithmetic operators must be applied to numbers! If types are not being checked, ill-typed inputs are coerced to default values. For example, if an arithmetic operator is applied to a non-number, the non-number is coerced to `0`.

For the purposes of this course, it's best for the prompt to look like "`ACL2 p>`", i.e., you should be in program mode with no type checking. But ACL2 typically starts up with the the prompt "`ACL2 !>`". So you should know how to switch between these different modes. The following session log illustrates how to do it.

[1] ACL2 is a *functional programming language* . Functional programming languages are also sometimes called "applicative" (because all you can do is apply functions) or "side-effect free" (because programs can only return values, not change things in the environment). There are other functional languages, Haskell being the most well known. But ACL2 is simpler than Haskell.

```
ACL2 !>(+ 2 3)
5
ACL2 !>(+ 2 nil)
ACL2 Error [Evaluation] in TOP-LEVEL:  The guard for the function call
(BINARY-+ X Y), which is (AND (ACL2-NUMBERP X) (ACL2-NUMBERP Y)), is
violated by the arguments in the call (BINARY-+ 2 NIL). ...
ACL2 !>(set-guard-checking nil)
Masking guard violations ...
ACL2 >(+ 2 nil)
2
ACL2 >(program)
ACL2 p>:set-guard-checking t
Turning guard checking on, value T.
ACL2 p!>(logic)
ACL2 !>
```

You can arrange for ACL2 to always start in the preferred mode by creating a file named `acl2-customization.lisp` on the directory where you start up your session. That file should have these two lines in it:

```
(program)
(set-guard-checking nil)
```

The following homework question can be answered by just evaluating the three expressions in the question, provided you're in the right mode.

• **QUESTION 14**: Learn to use ACL2 to define and evaluate simple programs. Below we define `(fact n)` to be the factorial function for natural numbers `n`, i.e., `(fact n)` is how we write $n!$ in ACL2. Then we define `(count-digs x d)` which counts how many times the digit $0 \leq d \leq 9$ occurs in the decimal expansion of `x`. Then we ask how many times the digit 7 appears in the decimal expansion of 157!. What is the answer?

```
(defun fact (n)
  (declare (xargs :mode :program))
  (if (zp n)
      1
      (* n (fact (- n 1)))))
(defun count-digs (d x)
  (declare (xargs :mode :program))
  (if (zp x)
      0
      (if (equal (mod x 10) d)
          (+ 1 (count-digs d (floor x 10)))
          (count-digs d (floor x 10)))))
(count-digs 7 (fact 157)) •
```

2.1 Evaluating Simple Expressions (Assignment 1 cont'd)

Instead of writing $\mathtt{n} \times (1 + \mathtt{n})$, ACL2 programmers write `(* n (+ 1 n))`. ACL2 uses *prefix syntax*, where the function symbol is written first and then the argument expressions are listed, without commas, and the whole thing is enclosed in parentheses.

Let `(`f a_1 ... a_k`)` be an ACL2 expression. Then f is a *function symbol* that takes k arguments and each of the a_i is an ACL2 expression that produces the corresponding argument value. Such an expression is called an *application* of the function symbol f to the (values of the) a_i. Sometimes we say `(`f a_1 ... a_k`)` is a *call* or *invocation* of f.

You may think of a *function* as a "machine" that takes inputs and produces an output. Almost all the functions you learned about in high school deal with numbers. For example:

$log_2(x)$	the base-2 logarithm of x	$log_2(32) = 5$
$sqrt(x)$	the square root of x usually written $\sqrt{x}$	$sqrt(9) = 3$
$tan(x)$	the tangent of x	$tan(60°) = 1.7317\ldots$
$floor(x)$	the largest integer i such that $i \leq x$ usually written $\lfloor x \rfloor$	$floor(12.379) = 12$
$gcd(x, y)$	greatest common divisor of x and y	$gcd(18, 27) = 9$

Technically, an ACL2 expression is called a "term." A *term* is a variable symbol, a constant, or a function call, written `(`f a_1 ... a_n`)`, where the a_i are terms.

Any ACL2 object can be used as a constant in a program. If c is some ACL2 object, and you wish to use it as a constant in a program, you may write `'`c or `(quote` c`)`. These are constant expressions that evaluate to c. Some constants, like numbers, need not be quoted. (By the way, to prevent `(quote` c`)` from being mistaken for a function call, `quote` may not be defined as a function.) The necessity of quoting certain kinds of objects will become clear later.

The number of arguments a function symbol "expects" is called its *arity.*

Calls can be nested. Suppose `f`, `g`, and `h` are function symbols of arity 1, 2, and 3 respectively. Then here are some ACL2 expressions and the corresponding expressions in traditional mathematical notation.

ACL2	*traditional notation*
`(f x)`	$f(x)$
`(f (g a b))`	$f(g(a, b))$
`(g (f a) b)`	$g(f(a), b)$
`(h a (f a) (g b c))`	$h(a, f(a), g(b, c))$

The following are not terms because they are ill-formed:

`(f x y)`	too many arguments
`(g x)`	too few arguments
`(f (g x))`	ill-formed interior "term"

A more familiar example of a term is `(* n (+ 1 n))`. This is a function call of the function symbol `*` with the two argument expressions `n` and `(+ 1 n)`. The expression `n` is just a variable symbol. The expression `(+ 1 n)` is a function call of the function symbol `+` on the constant expression `1` and the variable `n`.

• **Question 15**: Assume the following function symbols have the arities given. Pretend there are no other function symbols. You may assume that any contiguous sequence of alphabetic characters used in this question, e.g., `x` and `temp`, is a legal variable symbol. Mark with a check the legal terms below. Note that this doesn't require that you know what the terms *mean*!

All Known Functions (for now)

symbol	*arity*	*symbol*	*arity*
`if`	3	`+`	2
`cons`	2	`*`	2
`first`	1	`zp`	1
`rest`	1	`nth`	2
`consp`	1	`prime-factors`	1
`filter`	2		

Alleged Terms

```
 1.  (if x y z)
 2.  nth(3,filter(u,temp))
 3.  (if p (if q t nil) t)
 4.  (cons (first x) (rest x))
 5.  (cons 1 2 3)
 6.  3 + 4
 7.  (if (zp n)
         (first x)
         (nth (+ n -1) (rest x)))
 8.  (nth 3 (filter u v))
 9.  (filter (prime-factors x 7))
10.  (filter (prime-factors x) 7)
11.  (+ (* x x) (* -1 (* y y)))
12.  (if (x 7) (y 8) (t 9))
13.  (first rest)
14.  (x cons y)
15.  (if 1 2 3)
16.  (first 17)
17.  (if x y)
18.  (cons x (confabulate x y))
```

•

The *value* of a function call `(`f a_1 `...` a_k`)` is obtained by applying the function f to the values of the arguments.

So, for example, the value of `(* n (+ 1 n))` is obtained by applying the multiplication function (`*`) to the values of `n` and `(+ 1 n)`.

The value of a constant, e.g., `1`, is itself, of course.

But what about variables? The value of a variable must be given by the environment in which we're evaluating it. Think of the *environment* being a table that maps every variable symbol to some value. To get the value of a variable we look it up in the environment. If the value of variable v is some object c in the environment, we say v is *bound* to c. Variants on this terminology are "the *binding* of v is c in the environment" and "the environment *binds* v to c."

Suppose the value of `n` in the environment is `5`. Then the value of `(* n (+ 1 n))` is `30`. We will write

`(* n (+ 1 n))` $\Longrightarrow$ `30`

or more generally

$\alpha \Longrightarrow \beta$.

to mean the "expression α evaluates to the object β." Note that the environment used in the evaluation is unspecified in this notation and will be made clear by context.

We can work out mechanically how to evaluate `(* n (+ 1 n))`, just based on the three rules given above: how to evaluate a variable, a constant, and a function call.

- the value of the function call `(* n (+ 1 n))`,
  obtained by applying the multiplication function (`*`) to
  - the value of the variable `n`,
    obtained from the environment,
    is `5`, and
  - the value of the function call `(+ 1 n)`,
    obtained by applying the addition function (`+`) to
    - the value of the constant `1`,
      is `1`, and
    - the value of the variable `n`,
      obtained from the environment,
      is `5`,

    is `6`,

  is `30`.

Some students are helped by the following "trace" of the process:

```
(* n (+ 1 n))
=                                                    {environment}
(* 5 (+ 1 n))
=                                                    {environment}
(* 5 (+ 1 5))
=                                                    {evaluation of +}
(* 5 6)
=                                                    {evaluation of *}
30
```

The remarks in braces "{...}" are called *justifications*. When a justification is written between two formulas it explains how the formula above the remark is manipulated to produce the formula below the remark.

Because the value of an expression in a given environment never changes – there are no side-effects – it doesn't matter in what order we evaluate the sub-expressions. We could have written it this way:

```
(* n (+ 1 n))
=                                                    {environment}
(* n (+ 1 5))
=                                                    {evaluation of +}
(* n 6)
=                                                    {environment}
(* 5 6)
=                                                    {evaluation of *}
30.
```

This explanation of how to evaluate expressions leaves implicit the question of how to "apply" the multiplication and addition functions. For such familiar functions you probably understand. But how do we apply user defined functions? To explore that we have to learn how to define functions in ACL2.

2.2 Defining Functions (Assignment 1 cont'd)

You learned how to define simple functions in high school, by writing equations. For example, the area of a rectangle with width x and length y is given by:

$$area(x, y) = x \times y.$$

If you are given a specific rectangle, say, with width 5 and length 7, then you can compute $area(5, 7) = 5 \times 7 = 35$, by substituting 5 and 7 respectively for x and y in the definition of $area$ and doing the math.

Defining functions in ACL2 is no different. We could write

```
(defun area (x y) (* x y))
```

and then evaluate `(area 5 7)` and get `35`.

A definition takes the form

(defun f (v_1 ... v_n) β)

where f is the name of the new function, the v_i are distinct variable symbols, and β is some expression using those variables. The v_i are called the *formals* or *formal parameters* of the definition. The expression β is called the *body* of f.

A function definition in ACL2 is just an equation in math class. The definition above means

$$f(v_1, \ldots, v_n) = \beta$$

except, of course, we write it

$$(f\ \ v_1\ \ \ldots\ \ v_n)\ \ =\ \ \beta.$$

Consider a function call, e.g., (f a_1 ... a_n). Here, f is a function symbol of n arguments and the a_i are expressions. The a_i are called the *actual expressions* of the call. The actual expressions have values – we obtain them by evaluating the actual expressions. Those values are called the *actual values* or sometimes just the *actuals*. Note that when we refer to the "actuals" of a call, we are referring to their values, not the expressions. For example, in `(area (+ 2 3) (+ 3 4))`, the actuals are `5` and `7`, not "`(+ 2 3)`" and "`(+ 3 4)`".

To compute the value of a call of a defined function you evaluate the actual expressions to obtain the actual values (the "actuals"), you then create a new environment where the formals are bound to the actuals, and evaluate the body in that environment.

This is often abbreviated to "plug the actuals in for the formals and evaluate the body." You've done this thousands of times. It's like turning a crank.

Let's define a function, `foo`, that takes one argument, `n`. For the moment, let's just suppose that the input, `n`, is always a natural number (e.g., 0, 1, 2, ...). As indicated by its iconic name, `foo` is not meant to be an interesting function, just something concrete but arbitrary that we can discuss. The value of `foo` on `n` will be defined to be `n` $\times$ (1 + `n`), unless `n` is 0, in which case the value will be `1`.

Here is the ACL2 definition:

```
(defun foo (n)
 (if (zp n)
     1
     (* n (+ 1 n))))
```

The function `foo` has only one formal, `n`. The body of `foo` is just a call of the 3-argument function symbol `if`. The first argument of the `if`-expression is a call of the function symbol

`zp` on one argument, the variable `n`. The second and third arguments of the `if`-expression are familiar.

Note that except for the function symbols in the body of `foo`, there is nothing new here: expressions are constants, variables, or calls of functions on other expressions. To learn ACL2 all you have to do is learn how to write down the available constants (you've only seen numeric ones so far), the names and interpretation of a few primitive function symbols, and how to *compose* them, i.e., how to write nested calls of functions, to compute what you intend.

So what are `if` and `zp`? To answer that, I'll just tell you how to evaluate such expressions.

The value of `(if` α β γ`)` is either the value of β or the value of γ, depending on whether the value of α is true or false. Thus, `if` is an if-then-else expression. We call α the *test*, β the *true branch*, and γ the *false branch*.

Suppose the variable `p` is true in the current environment and suppose the variable `n` has the value 5. Then the value of `(if p (+ 1 n) k)` has the value `6` and it doesn't matter what value `k` has.

The value of `(zp` α`)` is either true or false. Think of `(zp` α`)` as testing whether the value of α is 0. In particular, if the value of α is `0`, the value of `(zp` α`)` is true. If the value of α is a natural number other than `0`, the value of `(zp` α`)` is false. Notice that I haven't told you what the value of `(zp` α`)` is when the value of α is not a natural number. I'll deal with that later.

Note how cumbersome it is to keep talking about the values of expressions. Here's the description of `if` again:

> *The value of* `(if` α β γ`)` is either *the value of* β or *the value of* γ, depending on whether *the value of* α is true or false.

When it is clear we are talking about *the value of* an expression, we don't keep repeating "the value of" – because the value of an expression always depends only on the values of the subexpressions, never on the expressions themselves.

In this style, the description of `if` is:

> *The value of* `(if` α β γ`)` is either β or γ, depending on whether α is true or false.

or even

> `(if` α β γ`)` is either β or γ, depending on whether α is true or false.

The description of `zp` is:

> (zp α) is true if α is `0` and is false if α is a natural number other than `0`. I haven't yet told you the value of (zp α) when α is not a natural number.

Sometimes we say `zp` is a *recognizer* for `0` in the sense that it returns true or false according to whether its (natural number) argument is `0`.

In general, a *recognizer* for a condition is a Boolean valued function that is true if the condition is true and false otherwise. Technically, `zp` is the recognizer for the condition that its argument is not a positive natural number. Often in ACL2, Boolean valued functions have names ending in "`p`," which stands for "predicate." The name "zp" thus stands for "zero predicate."

2.3 True, False, Apples, and Firetrucks (Assignment 2: 2 days)

In writing about `if` and `zp` above I repeatedly talked about something being "true" or "false." ACL2 has two constants `t` and `nil` called the *Booleans*. These are data objects, just like `0`, `1`, `2`, ... are. All these objects are different.

When I say some ACL2 value "is false" I mean that it is the object `nil`. When I say some ACL2 value "is true" I mean it is *not* the object `nil`. Note that I did not say that "x is true" means "x is `t`." The sentence "x is true" means "x is not `nil`."

Since `t` is not `nil`, then `t` is true. But `23` is not `nil` either. So `23` is true.

Many students get confused here! "If `t` is true and `23` is true, then `t` must be `23`, right?" Wrong! This confusion stems from misunderstanding what part of speech the word "true" is in sentences such as "`t` is true." I am using "true" as an **adjective**, not a **noun**.

Parse "`t` is true" the same way you would "the apple is red." "Red" is an adjective, not a noun. The apple is red. The firetruck is red. But that doesn't mean the apple is necessarily the firetruck. Just because `t` is true and `23` is true doesn't mean `t` is `23`!

Many ACL2 objects are true. In fact, *every* ACL2 object is true *except* `nil`!

Thus, every ACL2 object is either true or false, because every object is either not `nil` or is `nil`.

I'm using "false" as an adjective too, in such sentences as "x is false", but since there is only one false thing, you can think of false as a noun naming that thing. Thus, if x is false and y is false, then x is y, because they are both `nil`.

Reread the description of how to evaluate `(if` α β γ`)`. It's equivalent to this: Evaluate α. If the result is `nil`, evaluate and return γ. Otherwise, evaluate and return β.

• **Question 16**: Suppose `n` has value `5` in the environment. What is the value of `(if (+ n 1) (+ n 2) (+ n 3))`? •

• **Question 17**: Suppose `n` has value `5` in the environment and `x` has value `nil`. What is the value of `(if x (+ n 2) (+ n 3))`? •

It is very odd to use non-Boolean tests in `if`-expressions. We will do it rarely. But I'm trying to make a point: `if` doesn't "care" whether its argument is a Boolean or not. It tests against `nil` and goes one way or the other. In a test, any true value will do. That is, `(if`

`t` β γ), `(if 0` β γ), `(if 1` β γ), etc., all have the same value, namely the value of β.

When an ACL2 programmer wants a canonical true object, he or she uses the constant `t`.

Recall my description of `zp` above:

> `(zp` α`)` is true if α is `0` and is false if α is a natural number other than `0`. I haven't yet told you the value of `(zp` α`)` when α is not a natural number.

This description is incomplete in a way you probably didn't notice before: I didn't tell you *which* true thing `(zp` α`)` returns. In a sense, it doesn't matter: if all our uses of `zp` are in the tests of `if`s, then one true thing is as good as another. But just to be more precise:

> `(zp` α`)` is `t` if α is `0` and is `nil` if α is a natural number other than `0`. I haven't yet told you the value of `(zp` α`)` when α is not a natural number.

Suppose we have a term and we're only interested in whether it is true or false, but not in what object it computes when evaluated. We tend to call such terms *formulas*. A function whose value is Boolean – or one whose value is frequently tested as true or false – is often called a *predicate*. `Zp` is a predicate. But technically for us, "formula" is just another word for "term," "expression" is just another word for "term," and "predicate" is just another word for function.[2]

2.4 Evaluating Calls of Defined Functions (Assignment 2 cont'd)

Recall our definition of `foo`.

```
(defun foo (n)
 (if (zp n)
     1
     (* n (+ 1 n))))
```

What is the value of `(foo 5)`? More generally, how do we evaluate calls of `foo`?

To make a long story short, the value of `(foo 5)` is `30`.

If you fire up ACL2 and type the `defun` expression above you will define the `foo` function. Ignore the output produced unless it is some kind of error; no error should happen if you do it exactly right. Then if you type `(foo 5)` you will call `foo` on `5` and it will compute `30` and print it.

Here is how the ACL2 session looks on my machine after I've submitted the definition of `foo` and run `(foo 5)`.

[2]This is unconventional. In most treatments of logic, the truth values are not objects. Terms evaluate to objects and formulas are either "true" or "false." A predicate is like a function but takes objects as input and is either "true" or "false". But in Lisp, the usage here is quite common.

```
ACL2 p>(defun foo (n)
 (if (zp n)
     1
     (* n (+ 1 n))))
Summary
Form:    ( DEFUN FOO ...)
Rules:  NIL
Time:    0.01 seconds (prove:  0.00, print:  0.00, other:  0.01)
 FOO
ACL2 p>(FOO 5)
30
```

For the purposes of this course, you can just ignore the `Summary`. All that matters is that ACL2 accepted the definition. If it hadn't accepted it, it would have printed an error message.

Note also that ACL2 is not *case sensitive* when it comes to names of functions, variables, and constants. For example, `foo`, `Foo`, and `FOO` are all ways of writing the same name. `NIL`, `nil`, and `NiL` are the same.

While it is helpful to use ACL2 to evaluate function calls, you *must* know how to do it yourself.

Here is the rule, again. Suppose we have defined `(defun` f `(`v_1 ... v_k`)` β`)`. Then to evaluate `(`f a_1 ... a_k`)` you evaluate the a_i to obtain some values, c_i. Then you create an environment where the formals of f, namely v_1, ..., v_k, have the values c_1, ..., c_k, respectively, and you evaluate β in that environment and return the result. The environment you created for the evaluation of β may then be discarded.

Let's evaluate `(foo (+ n 30))` in an environment, *env*, where `n` is `5`. The expression is a call of the defined function `foo` on the expression `(+ n 30)`. Evaluating the argument expression in *env* produces `35`. Next, we create a new environment, env', where the local of `foo`, which happens to also be named `n`, has the value `35`. Finally, evaluate the body of `foo` in env' and return the result.

The body of `foo` is

```
(if (zp n)
    1
    (* n (+ 1 n)))
```

and `n` has the value of `35` in env'. The test of the `if`-expression, `(zp n)`, evaluates to false (`nil`), because the value of `n` in env' is a positive natural. So the value of the `if`-expression is the value of its false branch, `(* n (+ 1 n))`. You should be able to work out that the value of the false branch in env' is `1260`.

Thus, the value of `(foo (+ n 30))` in *env* is `1260`.

Note that we don't need env' after we've finished the evaluation of the body of `foo`.

Note we don't "modify" *env* to get env'. For example, consider the expression `(+ (foo (+ n 30)) n)`. Note how this expression uses `n` again after the call of `foo`. The value of this expression in *env* is the sum of `1260` and 5. Some students might imagine we "modify" *env* when we call `foo` and permanently set the value of `n` to `35`. If so, when we return from `foo` and continue evaluating we would get `35` as the (new) value of the second `n`. So the (wrong) answer would be `1260` + `35` or `1295`. That's wrong! *Calling a function never changes anything in the caller's environment.*

Finally, note that the *only* variable symbols in env' are the formals of `foo`. This would raise problems if we encountered a variable in the body of `foo` that the environment didn't mention. But it will turn out that the only variables in the body of a `defun` are the formals.

Here is an evaluation of `(foo (+ n 30))` written in equational form. Note that the starting environment is *env* where `n` has value `5`.

```
(foo (+ n 30))
=                                        {environment env}
(foo (+ 5 30))
=                                        {evaluation of +}
(foo 35)
=                 {definition of foo (in new environment env')}
(if (zp 35)
    1
    (* 35 (+ 1 35)))
=                                  {evaluation of (zp 35)}
(if nil
    1
    (* 35 (+ 1 35)))
=                                        {evaluation of if}
(* 35 (+ 1 35))
=                                        {evaluation of +}
(* 35 36)
=                                        {evaluation of *}
1260
```

The step above justified "by definition of `foo` ..." is actually quite subtle. Note that I replaced the call of `foo`, namely `(foo 35)`, by the body of `foo` *in which I'd substituted* the actual, `35`, for the symbolic formal, `n`.

This is a way to avoid talking explicitly about environments: To evaluate a function call, evaluate the arguments, plug the actual values in for the formals in the body, and evaluate the instantiated body. "Plugging in the actual values" is sometimes called *instantiating* the body with the actuals.

Warning: There is a subtle problem with this short-cut description. When we evaluate the instantiated body we re-evaluate the actuals each time we encounter them in the body. In this case, the only actual is the constant `35` and every time we evaluate it we get `35`. But just imagine there is some object c that when evaluated produces something, d, different from c!

Right now you've only seen numbers and Booleans and they always evaluate to themselves. But if c were the value of an actual expression, then upon plugging c into the body and evaluating it, we'd get d in the places where the environment would have produced the value c. Clearly, the plug-in-the-actuals scheme has to be understood in a way that avoids this confusion. We'll describe later. As long as the actuals are numbers and Booleans, the plug-in-the-actuals scheme works perfectly.

2.5 Recursion (Assignment 2 cont'd)

That's all there is to the ACL2 programming language, except that I must tell you the various kinds of constants (besides the natural numbers and the two Booleans) and the primitive functions (besides `if`, `zp`, `+`, and `*`).

Before we dive into those things, it is helpful to illustrate the power of the language we've got. Let's define the factorial function, which is often introduced informally as

$$n! = n \times (n-1) \times \ldots \times 2 \times 1$$

Many students who have focused on programming in such languages as Java will assume it's impossible to define factorial because we don't have a `while` statement. That's wrong. We have function call, and that is *much more powerful*. What makes function call so powerful is that we can define functions recursively. A definition of f is *recursive* if the body of f calls f.

We'll define `fact` to take one argument, `n`, which we'll assume is a natural number. Here's how you define it in ACL2.

```
(defun fact (n)
  (if (zp n)
      1
      (* n (fact (+ n -1)))))
```

So how does computation work in ACL2? Let's figure out the value of `(fact 3)`, step by step.

```
(fact 3)
=                                          {definition of fact}
(if (zp 3)
    1
    (* 3 (fact (+ 3 -1))))
=                                {since (zp 3) is nil (3 is not 0)}
(if nil
    1
    (* 3 (fact (+ 3 -1))))
=                                          {evaluation of if}
```

```
(* 3 (fact (+ 3 -1)))
=                                              {evaluation of +}
(* 3 (fact 2))
=                                              {definition of fact}
(* 3 (if (zp 2)
         1
         (* 2 (fact (+ 2 -1)))))
=                                   {since (zp 2) is nil (2 is not 0)}
(* 3 (if nil
         1
         (* 2 (fact (+ 2 -1)))))
=                                              {evaluation of if}
(* 3 (* 2 (fact (+ 2 -1))))
=                                              {evaluation of +}
(* 3 (* 2 (fact 1)))
=                                              {definition of fact}
(* 3 (* 2 (if (zp 1)
              1
              (* 1 (fact (+ 1 -1))))))
=                                   {since (zp 1) is nil (1 is not 0)}
(* 3 (* 2 (if nil
              1
              (* 1 (fact (+ 1 -1))))))
=                                              {evaluation of if}
(* 3 (* 2 (* 1 (fact (+ 1 -1)))))
=                                              {evaluation of +}
(* 3 (* 2 (* 1 (fact 0))))
=                                              {definition of fact}
(* 3 (* 2 (* 1 (if (zp 0)
                   1
                   (* 0 (fact (+ 0 -1)))))))
=                                              {since (zp 0) is t}
(* 3 (* 2 (* 1 (if t
                   1
                   (* 0 (fact (+ 0 -1)))))))
=                                              {evaluation of if}
(* 3 (* 2 (* 1 1)))
=                                     {evaluation of * three times}
6
```

To compute with ACL2, all you have to understand is how to evaluate constants, variables, calls of function symbols.

2.6 Practice with Definitions (Assignment 2 cont'd)

In each of the problems in this section, you are to fill in the blanks to make the function defined compute the answer described.

The only function symbols you may use in the blanks for a given question are `if`, `zp`, `+`, and the functions defined in previous questions of this section. Note that I don't want you to use `*`. So, for example, below I define `add1` and `sub1`, and you may use them in your answers to subsequent questions.

In all cases below, you may assume we'll only call the functions with natural numbers as arguments. That is, I don't care what your definition does if the function is applied to an argument that is not a natural number.

I work a couple of problems for you to teach you how to define recursive functions.

• **Question 18**: Fill in the blank to define `add1` to return a number 1 greater than its argument.

(defun add1 (n) ___α___)

Show tests of your definition on these examples (at least):

(add1 0) $\Longrightarrow$ 1
(add1 1) $\Longrightarrow$ 2
(add1 2) $\Longrightarrow$ 3
(add1 17) $\Longrightarrow$ 18 •

My Answer:

α : `(+ 1 n)`

Thus, I'm defining

```
(defun add1 (n) (+ 1 n))
```

and `add1` returns a number 1 greater than its argument; for example, `(add1 7)` is `8`. I don't care what `(add1 nil)` is because we are assuming `n` is a natural number.

To answer this question on a homework assignment you should turn in an extract of an ACL2 session that looks like this:

```
ACL2 p> (defun add1 (n) (+ 1 n))
Summary
Form:    ( DEFUN ADD1 ...)
Rules:  NIL
Warnings:    None
Time:    0.01 seconds (prove:  0.00, print:  0.00, other:  0.01)
 LEN
ACL2 p>(add1 0)
```

```
1
ACL2 p>(add1 1)
2
ACL2 p>(add1 2)
3
ACL2 p>(add1 17)
18
```

Note that you don't have to tell me what α is, just show me the final definition and some executions of your function.

• **Question 19**: Fill in the blank to define `sub1` to return a number 1 less than its argument, unless its argument is `0` in which case it returns `0`.

```
(defun sub1 (n)
  (if __α__
      0
      (+ __β__ n)))
```

Show tests of your definition on these examples (at least):

```
(sub1 0) ⟹ 0
(sub1 1) ⟹ 0
(sub1 2) ⟹ 1
(sub1 17) ⟹ 16   •
```

My Answer:

α : `(zp n)`
β : `-1`

Thus, I'm defining

```
(defun sub1 (n)
  (if (zp n)
      0
      (+ -1 n)))
```

Thus, `(sub1 7)` is `6` and `(sub1 0)` is `0`.

Now here is an example requiring recursion. You might have to *think* about how this function works.

• **Question 20**: Fill in the blank to define `double` so that it doubles its argument.

```
(defun double (n)
  (if (zp n)
      __α__
      (+ __β__
```

```
        (double (sub1 n))))) •
```

Show tests of your definition on these examples (at least):

```
(double 0) ⟹ 0
(double 1) ⟹ 2
(double 3) ⟹ 6
(double 17) ⟹ 34  •
```

Ok, this is tricky. I'm making you fill in the blanks and I've written some "glue code" around the blanks. The game we're playing is that you have to live with my glue code and make `double` double `n`.

So let's think. The test asks whether `n` is `0` and α is the answer when it is. So what's α? Think! What's the result of doubling `0`, i.e., 2×0?

Moving on, if the test is false, we have to add β to the value of `(double (sub1 n))`. First, what is the value of `(double (sub1 n))`? Well, if we do our job right, it will be the double of `(sub1 n)`, i.e., $2 \times (n-1)$ or $2n-2$. So what do we add to that to get the double of `n`?

My Answer:

α : `0`
β : `2`

So I've defined

```
(defun double (n)
  (if (zp n)
      0
      (+ 2 (double (sub1 n)))))
```

Let's test it on 3.

```
(double 3)
=                    {def double, eval of (zp 3), (if nil ...), and (sub1 3)}
(+ 2 (double 2))
=                    {def double, eval of (zp 2), (if nil ...), and (sub1 2)}
(+ 2 (+ 2 (double 1)))
=                    {def double, eval of (zp 1), (if nil ...), and (sub1 1)}
(+ 2 (+ 2 (+ 2 (double 0))))
=                    {def double, eval of (zp 0) and (if t ...)}
(+ 2 (+ 2 (+ 2 0)))
=                    {arithmetic}
6
```

So it seems to work on `3` at least!

Now it's your turn. Remember, you can only use the functions specified at the beginning of this section (including the ones I've defined above). You can use any function defined

by a previous question, even if you are not required to answer that previous question on a homework. Note that to answer these with ACL2 you will have to add my definitions of `add1`, `sub1`, and `double` above to your session; you'll also have to define any functions from previous questions that you use. I don't care what your functions do if applied to arguments that are not natural numbers. Finally, as always with programming, there are many different correct answers.

• **Question 21**: Fill in the blanks to define `flip` so that `(flip 0)` is `1`, `(flip 1)` is `0`, and on any other natural numbers n, `(flip` n`)` is `nil`.

```
(defun flip (n)
  (if (zp n)
      ___α___
      (if ___β___
          ___γ___
          nil)))
```

Show tests of your definition on these examples (at least):

```
(flip 0) ⟹ 1
(flip 1) ⟹ 0
(flip 7) ⟹ NIL
```

Remember, `NIL` and `nil` are just two different ways to write the same symbol. •

• **Question 22**: Fill in the blank to define `equal-1` so that `(equal-1 n)` is `t` if `n` is `1` and `nil` if `n` is anything else. You may assume `n` is a natural number. Remember, you can only use the functions described at the beginning of this section.

```
(defun equal-1 (n) ___α___)
```

Show tests of your definition on these examples (at least):

```
(equal-1 0) ⟹ NIL
(equal-1 1) ⟹ T
(equal-1 2) ⟹ NIL
(equal-1 3) ⟹ NIL
```

Note: Of course, ACL2 has a primitive for asking whether two objects are the same. But the game we're playing here is to challenge you to use just the functions you've already seen to answer the question whether `n` is `1`. •

• **QUESTION 23**: Fill in the blanks to define `plus` so that it returns the sum of the natural numbers `n` and `m`. Turn in an ACL2 transcript like that shown in my discussion of Question 19 (page 19): the `defun` and the results of evaluating examples, including those below.

```
(defun plus (n m)
```

```
  (if (zp n)
      ___α___
      (___β___
         (plus (___γ___ n) m))))
```

Note that β and γ above must be function symbols and each is of arity 1. Show tests of your definition on these examples (at least):

(plus 2 2) $\Longrightarrow$ 4
(plus 7 5) $\Longrightarrow$ 12
(plus 0 3) $\Longrightarrow$ 3 •

• **QUESTION 24**: Fill in the blanks to define `times` so that it returns the product of the natural numbers `n` and `m`. Turn in an ACL2 transcript like that shown in my discussion of Question 19 (page 19): the `defun` and the results of evaluating examples, including those below.

```
(defun times (n m)
  (if (zp n)
      ___α___
      (___β___
         ___δ___
         (times (___γ___ n) m))))
```

Note that β must be a function symbol of arity 2 and γ must be a function symbol of arity 1. Show tests of your definition on these examples (at least):

(times 7 3) $\Longrightarrow$ 21
(times 3 5) $\Longrightarrow$ 15
(times 1 5) $\Longrightarrow$ 5
(times 0 3) $\Longrightarrow$ 0 •

• **QUESTION 25**: Fill in the blanks to define `power` so that it returns `n` raised to the power `m`, where `n` and `m` are naturals. Note that by convention, 0^0 is 1. Turn in an ACL2 transcript like that shown in my discussion of Question 19 (page 19): the `defun` and the results of evaluating examples, including those below.

```
(defun power (n m)
  (if (zp m)
      ___α___
      (___β___
         ___δ___
         (power n (___γ___ m)))))
```

Note that β must be a function symbol of arity 2 and γ must be a function symbol of arity 1. Show tests of your definition on these examples (at least):

```
(power 2 3) ⟹ 8
(power 2 4) ⟹ 16
(power 2 5) ⟹ 32
(power 5 2) ⟹ 25
(power 0 0) ⟹ 1
(power 0 1) ⟹ 0
```
•

• **Question 26**: Fill in the blanks to define `even-natp` so that it returns `t` if its argument is an even natural number and `nil` if its argument is an odd natural number. Remember that you may assume `n` is a natural number.

```
(defun even-natp (n)
  (if (zp n)
      __α__
      (if (equal-1 n)
          __β__
          (even-natp __γ__))))
```

Show tests of your definition on these examples (at least):

```
(even-natp 0) ⟹ T
(even-natp 1) ⟹ NIL
(even-natp 2) ⟹ T
(even-natp 5) ⟹ NIL
(even-natp 18) ⟹ T
(even-natp 25) ⟹ NIL
```

Hint: In past problems you've always decremented `n` by `1` but you don't need to always decrement by `1`! •

• **Question 27**: Fill in the blanks to define `even-natp1` so that it computes the same thing that `even-natp` computes, but does it in a different way.

```
(defun even-natp1 (n)
  (if (zp n)
      __α__
      (if (even-natp1 (sub1 n))
          __β__
          __γ__)))
```

Show tests of your definition on these examples (at least):

```
(even-natp1 0) ⟹ T
(even-natp1 1) ⟹ NIL
(even-natp1 2) ⟹ T
(even-natp1 5) ⟹ NIL
(even-natp1 18) ⟹ T
```

```
(even-natp1 25) ⟹ NIL  •
```

• **Question 28**: Fill in the blanks to define `half` so that it returns the floor of `n`/2.

```
(defun half (n)
  (if (zp n)
      __α__
      (if (equal-1 n)
          __β__
          (+ 1 (half __γ__))))) •
```

Show tests of your definition on these examples (at least):

```
(half 0) ⟹ 0
(half 1) ⟹ 0
(half 2) ⟹ 1
(half 5) ⟹ 2
(half 18) ⟹ 9
(half 25) ⟹ 12  •
```

• **QUESTION 29**: Fill in the blanks to define `lessp` so that it returns `t` if its first argument is strictly smaller than its second and `nil` otherwise. Turn in an ACL2 transcript like that shown in my discussion of Question 19 (page 19): the `defun` and the results of evaluating examples, including those below.

```
(defun lessp (n m)
  (if (zp n)
      __α__
      (if (zp m)
          __β__
          (lessp (sub1 n) (sub1 m))))) •
```

Show tests of your definition on these examples (at least):

```
(lessp 1 2) ⟹ T
(lessp 2 1) ⟹ NIL
(lessp 3 5) ⟹ T
(lessp 17 13) ⟹ NIL
```

• **Question 30**: Fill in the blanks to define `equal-natsp` to return `t` if its two arguments are equal and `nil` otherwise. Both arguments may be assumed to be naturals.

```
(defun equal-natsp (n m)
  (if (zp n)
      __α__
```

```
      (if (zp m)
          β
          (equal-natsp (sub1 n) (sub1 m))))) •
```

Show tests of your definition on these examples (at least):

```
(equal-natsp 0 0) ⟹ T
(equal-natsp 1 1) ⟹ T
(equal-natsp 2 2) ⟹ T
(equal-natsp 17 17) ⟹ T
(equal-natsp 0 2) ⟹ NIL
(equal-natsp 2 0) ⟹ NIL
(equal-natsp 2 1) ⟹ NIL
(equal-natsp 17 13) ⟹ NIL
(equal-natsp 17 18) ⟹ NIL
```

Ok, here is a different kind of question. I'll answer a couple to show you what I want and how to do it. By the way, we're going to be defining a lot of little functions and I don't want to give them each an interesting name. So I'll sometimes ask you to define a function with a name like `fn-q`*i*, where *i* is the number of the Question.

• **Question 31**: What value is returned by `(fn-q31 5 7)` if `fn-q31` is defined this way:

```
(defun fn-q31 (n m)
  (if (zp m)
      n
      (add1 (fn-q31 n (sub1 m))))) •
```

My Answer: The easiest way to answer this kind of question is to use ACL2 to define `fn-q31` and then to evaluate `(fn-q31 5 7)`. If you do that, you learn that the answer is `12`.

• **Question 32**: For the `fn-q31` defined in the previous question, what is the value of `(fn-q31 x y)` when `x` and `y` are natural numbers? Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

My Answer: By using ACL2 to evaluate different calls of `fn-q31` I can learn:

```
(fn-q31 0 7) ⟹ 7
(fn-q31 1 7) ⟹ 8
(fn-q31 2 7) ⟹ 9
(fn-q31 3 7) ⟹ 10
```

So I can guess that `(fn-q31 x y)` adds `x` and `y`. A correct answer is thus `(+ x y)`. Another correct answer is `(plus x y)`, since `plus` has been mentioned in this section and also adds

its arguments.

• **QUESTION 33**: For the definition of `fn-q33` below, what is the value of `(fn-q33 x y)` when `x` and `y` are natural numbers?

```
(defun fn-q33 (n m)
  (if (zp n)
      m
      (fn-q33 (sub1 n) (add1 m))))
```

Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 34**: For the definition of `fn-q34` below, what is the value of `(fn-q34 x)` when `x` is a natural number?

```
(defun fn-q34 (n)
  (if (zp n)
      0
      (+ n (fn-q34 (sub1 n)))))
```

Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 35**: For the definition of `fn-q35` below, what is the value of `(fn-q35 x y 0)` when `x` and `y` are natural numbers?

```
(defun fn-q35 (n m a)
  (if (zp n)
      a
      (fn-q35 (sub1 n) m (+ m a))))
```

Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 36**: For the definition of `fn-q36` below, what is the value of `(fn-q36 x y)` when `x` and `y` are natural numbers?

```
(defun fn-q36 (n m)
  (if (zp m)
      0
      (+ n (fn-q36 n (sub1 m)))))
```

Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 37**: For the definition of `fn-q37` below, what is the value of `(fn-q37 x y 1)` when `x` and `y` are natural numbers?

```
(defun fn-q37 (n m a)
  (if (zp m)
      a
    (if (even-natp m)
        (fn-q37 (times n n) (half m) a)
      (fn-q37 n (sub1 m) (* n a)))))
```

Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 38**: For the definition of `fn-q37` used in the previous problem, what is the value of `(fn-q37 x y a)` when `x`, `y`, and `a` are natural numbers? Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

In the questions below, use the following definition of `uaf`.

```
(defun uaf (k n m)
  (if (zp k)
      (+ 1 n)
      (if (zp m)
          (if (equal k 2)
              0
              (if (equal k 3)
                  1
                  n))
          (uaf (- k 1)
               (uaf k n (- m 1))
               n))))
```

Warning: If you play with this function in ACL2 you are advised to stick to *very* small inputs, like `(uaf 2 3 2)`. Believe it or not, for quite modest inputs, the amount of time this function takes to compute and the numbers it creates are so large that no real computer could compute using the definition above. For example, `(uaf 4 3 5)` has 116 digits:

87189642485960958202911070585860771696964072404731750085525219437990967093723439943475549906831683116791055225665627

• **Question 39**: For the definition of `uaf` above, what is the value of `(uaf 1 3 5)`? •

• **Question 40**: What is the value of `(uaf 1 n m)` when `n` and `m` are natural numbers? Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 41**: For the definition of `uaf` above, what is the value of `(uaf 2 3 5)`? •

• **Question 42**: What is the value of `(uaf 2 n m)` when `n` and `m` are natural numbers? Answer with an ACL2 arithmetic expression involving the two variables, constants, and the

functions already mentioned in this section. •

• **Question 43**: For the definition of `uaf` above, what is the value of `(uaf 3 3 5)`? •

• **Question 44**: What is the value of `(uaf 3 n m)` when `n` and `m` are natural numbers? Answer with an ACL2 arithmetic expression involving the two variables, constants, and the functions already mentioned in this section. •

• **Question 45**: For the definition of `uaf` above, write an arithmetic expression in terms of small constants that says what `(uaf 4 3 5)` is equal to. I do *not* want you to repeat to me the 116 digit number shown above, which is indeed the value of `(uaf 4 3 5)`! To answer this question you might want to answer the next question first. •

• **Question 46**: Describe the value of `(uaf 4 n m)` when `n` and `m` are natural numbers. There are only two ways to answer this question. One would be to write an English description, possibly involving elipses ("...") . The other would be to define an ACL2 function of `n` and `m` that computes the same thing in more familiar terms. Either answer is acceptable. •

2.7 Lessons from the Practice (Assignment 3: 5 days)

One lesson from the exercises above is that functions can be defined many different ways. For example, `plus` in Question 23, `fn-q31` in Question 31, `fn-q33` in Question 33, `fn-q35` in Question 35, and `uaf` in Question 38 are all different ways to add two naturals.

In addition, the `defun`s above should convince you that you can define all of the basic arithmetic functions if you just know how to add and subtract `1`, how to recognize `0`, and how to use recursion. The definitions above are not very efficient. For example, to compute (power 10 3) we "build" 1,000 by a thousand additions of 1. But we are not concerned with efficiency. We are concerned with using the language we have to say what we need to say.

The problems have been posed as *programming* problems: think of a computational process to construct a certain number. But the answers have been expressed in a style that should feel *mathematical*. Any computational problem can be reduced to a mathematical problem. Sometimes your thought processes will be clearer if you eliminate the clutter of conventional programming languages and think mathematically. Often times good programmers conceive the solution in these terms and then just "code it up." The creativity does not lie in the coding! The creativity is in the algorithm and the algorithm can be expressed mathematically.

Another lesson from Section 2.6 is that there are many different algorithms to solve a given problem. For example, we saw several ways to add, to multiply, to exponentiate, to determine if a number is even, etc. But *how do you know* two different algorithms produce identical results? How do you know that `even-natp` and `even-natp1` return the same answer? Because we've expressed them mathematically we can prove they're the same using mathematics. That is something you'll learn to do in this course.

You will often be confronted by hideously complicated computational problems. Knowing

that you can express them mathematically and use mathematical tools – substitution of equals for equals, rearranging terms, etc. – to analyze them will empower you; the ability to use mathematical tools to solve computational problems is the difference between a coder and a computer scientist.

2.8 About Recursion (Assignment 3 cont'd)

Recursion is one of the most powerful ideas in computing, both theoretically and practically. It is also one of the most confusing things for new students to grasp.

To define a function f to solve some "big" problem you think of a smaller problem of the same kind, solve it by calling f, and then convert the solution of the smaller problem into a solution of the bigger one. By repeatedly reducing the size of the problem you eventually arrive at one you inherently know how to solve (the *base case*).

Every new programmer protests: "How can I call f when I haven't defined it yet?" or "How can I call f when I don't know how it works?" Defining a function recursively takes a leap of faith. When you finish defining it, f will be defined! You may not know how f works when you first write that recursive call, but you must assume that it *correctly solves the smaller problem* and get on with the task of converting the solution of the small problem to the solution of the big one.

So to define a function to solve some problem recursively you have to think of several creative things at once.

- a small version of the problem you can solve immediately
- a way to reduce any "big" problem to a smaller one so that doing this repeatedly reduces any problem to the "small version" you know how to solve, and
- how to create a solution for a "big" problem from a solution for its smaller one.

Here's a function to triple a number.

```
(defun triple (n)
  (if (zp n)
      0
      (+ 3 (triple (sub1 n)))))
```

Consider the three creative points above. The "small version" of the problem we know how to solve is how to triple `0`. The answer is `0`. This is called the *base case* of the recursion. We use `if` and `(zp n)` to determine if we are in this special case. If we're not, we're in the *recursive case* where we must reduce a "big" problem to a smaller one. But now we know $n > 0$ and so $n - 1$ is a smaller problem. Just *assume* that calling `triple` on $n - 1$ returns the correct answer to that question. Call the answer a. Can we find a way to convert the answer for $n - 1$ to the answer for n? Yes! Since we assume the recursive call answers

correctly, we know $a = 3 * (n - 1) = 3n - 3$. So to get $3n$, all we do is add 3. So we've solved all three creative problems and have a function that triples any natural.

You don't have to reduce the problem by just `1` every time. We saw in `even-natp1` an example where we reduce the problem by `2`. To answer "is `7` even?" we ask "is `5` even?". This is a case where we picked a smaller problem with the same answer as the bigger one. We answer "is `5` even?" by asking "is `3` even?". We answer that by asking "is `1` even?" Now if we continued and converted that to "is `-1` even?" we would skip over our base case and go running forever down the negatives: "is `-3` even?", "is `-5` even?", "is `-7` even?" Clearly, if we plan is to repeatedly reduce the problem by `2`, then we have to stop when we get to a problem whose size is smaller than `2`! That means we have two base cases: `0` and `1`. We know how to answer "is `0` even?" (yes) and "is `1` even?" (no). So we're done.

Believe it or not, ACL2 is an incredibly powerful programming language. In program mode it is *Turing complete* meaning that it can compute any computable function! No programming language can do more.

2.9 Comments and Displaying Code (Assignment 3 cont'd)

ACL2 code can be commented and displayed in various ways. Since we are allowed to quote numeric constants (even though we don't need to) we could write `foo` this way

```
(defun foo (n)
 (if (zp n)
     '1
     (* n (+ '1 n))))
```

without changing its meaning at all.

We could have also written it this way:

```
(defun foo (n)
 (if (zp n)
     (quote 1)
     (* n (+ (quote 1) n))))
```

"Whitespace" can be freely interspersed between ACL2 expressions. *Whitespace* refers to newlines, spaces, and tabs. Thus, we could write the definition of `foo` in any of these ways if we wanted:

```
(defun foo (n)
 (if (zp n)
     1
     (* n
        (+ 1
           n))))
```

```
(defun foo (n)
 (if (zp n) 1 (* n
     (+ 1 n))))
```

```
(defun foo (n)
 (if (zp n) 1 (* n (+ 1 n))))
```

```
(defun foo (n) (if (zp n) 1 (* n (+ 1 n))))
```

I discourage you from using the second form above because it makes it unclear what the false branch of the `if`-expression is.

Learn to "pretty print" your code. It is virtually impossible for me or you to read stuff like this:

```
(defun uaf (k n m) (if (zp k) (+ 1 n) (if (zp m) (if (equal k 2) 0 (if (equal
k 3) 1 n))
(uaf (- k 1) (uaf k n (- m 1)) n))))
```

The first rule is never write beyond the 80th column. This causes "wrap-around" and makes things very hard to read.

Always arrange your text so that it makes the structure of your thinking clear. I would display the definition above this way:

```
(defun uaf (k n m)
  (if (zp k)
      (+ 1 n)
      (if (zp m)
          (if (equal k 2)
              0
              (if (equal k 3)
                  1
                  n))
          (uaf (- k 1)
               (uaf k n (- m 1))
               n))))
```

This advice will help you avoid mistakes! The point is not to squeeze the text into the space allowed. The point is to record and communicate your ideas. You will read your stuff more often than anybody else does. I guarantee that if you learn to express yourself clearly to others you will produce work that you understand better and that is more often correct. If you carry these habits into your career as a programmer and designer, you will produce

designs that you can understand, maintain, and modify more reliably.

Here are three heuristic guidelines. (i) Look for ways to break up a line if it contains more than about 25 non-blank characters and try never to produce a line with more than about 40 characters. (ii) If you have to break up any argument, first indent that argument and all of its peers. (iii) Indent every argument equally.[3]

For example, while I would write this:

```
(if (zp k) (+ 1 n) (foo (+ 2 m) k m))
```

because the whole `if`-expression fits in 37 characters. But if I had to write something slightly longer, like

```
(if (zp k) (+ 1 n) (foo (+ 2 n) (+ 1 (* (+ k k) j)) k))
```

I would break it up. One way to break it up would be:

```
(if (zp k) (+ 1 n) (foo (+ 2 n)
                        (+ 1 (* (+ k k) j)) k))
```

But this violates guideline (ii) because I broke up the third argument of the `if`-expression (the `foo`-expression) but I tried to print its peers on one line, pushing the poor `foo`-expression out to the right. Trying to solve the problem by squeezing the `foo`-expression produces an ugly result. Instead, I could try to bring the `foo`-expression all the way in so that lines up with its first peer, the `(zp k)`.

```
(if (zp k) (+ 1 n)
    (foo (+ 2 n) (+ 1 (* (+ k k) j)) k)).
```

But that would violate guideline (iii) because the arguments to the `if` would not be indented equally. So I would indent the `(+ 1 n)` even though it "fits" where it is.

```
(if (zp k)
    (+ 1 n)
    (foo (+ 2 n) (+ 1 (* (+ k k) j)) k)).
```

If I felt the `foo`-expression was still too hard to read – what is its last argument? – I would indent all of its arguments:

```
(if (zp k)
    (+ 1 n)
    (foo (+ 2 n)
         (+ 1 (* (+ k k) j))
         k)).
```

[3] `Defun` is an exception: it is conventional to write the "`defun`", function name and list of formals on one line and then indent the body a space or two.

The point is *not* just to make the text fit in 80 columns! The point is to make the structure of the formula clear. The indentation tells me which expressions belong to which function call.

If you turn in stuff like this, the graders and I simply won't read it:

```
(defun uaf (k n m)
  (if (zp k)
      (+ 1 n)
      (if (zp m)
      (if (equal k 2)
      0
      (if (equal k 3)
          1
          n))
      (uaf (- k 1)
        (uaf k n (- m 1))
           n)))).
```

If you can't be bothered to make your thoughts clear, we can't be bothered to check them. Remember: use the format of your code to *communicate* to the reader. Your job is to bring the reader along with you. Most of the time *you will be your own reader* and this advice helps you more than anybody else.

Finally, it is possible to add comments to ACL2 code. Comments are delimited on the left by a semi-colon and on the right by the end of the line. So we could add comments to `foo` this way:

```
(defun foo (n)        ; This is a trivial
                      ; example of a definition.

 (if (zp n)           ; if n is 0
     1                ;    return 1
     (* n (+ 1 n))))  ;    else, n*(1+n)
```

Warning: You will get a 0 for any answer that violates the following rules. Every "definition" must be syntactically well-formed. Nothing should extend beyond the 80th column – lines should not be truncated or wrap around when viewed in a window 80 characters wide. Finally, if the grader feels that insufficient effort has been made to display a formula sensibly, he or she is free to assign a 0 for that answer even if it is otherwise correct!

Warning: If you play with ACL2 or Lisp, *never write a comma*! The program that reads your type-in does something very special on commas and you don't want to go there! There are a few other "bad characters," among them hash mark "`#`" and dot "`.`" but they are occasionally used in certain situations. Also, when writing strings, `"Hello, World!"`, the rules are relaxed. But naked commas, hash marks, and dots can cause trouble. So don't type them unless you know what you're doing! If you type one of these characters to ACL2 and don't know what you're doing, it may print a scary warning including something like

```
************ ABORTING from raw Lisp ***********
```

Just ignore it and correct your input.

2.10 The Five Types of Objects (Assignment 3 cont'd)

You have only seen two kinds of objects so far: numbers and Booleans. ACL2 actually provides five kinds or "types" of objects.

- numbers: ACL2 provides the integers and other kinds of numbers. We will only use the integers here, especially the naturals, which are a subset of the integers.
- characters: ACL2 provides objects that represent characters like "upper case A" and "space", but we won't use them.
- strings: A string is a finite sequence of characters delimited by the string quote (`"`) character, e.g., `"Hello World!"`. You may see a few strings in my code but we won't make much use of them.
- symbols: A symbol may be thought of as an object representing a *name*. `Monday`, `Green`, `public`, and `int` are all symbols that might be used in the data we manipulate. For our purposes, case is unimportant: `fact`, `Fact`, and `FACT` all denote the same symbol. The Booleans, `t` and `nil`, are just symbols. `()` is just another way you can write the symbol `nil`, although ACL2 will never print `nil` that way. Other symbols involving "unusual" characters like spaces, parentheses, etc., must be specially delimited but we do not use such symbols here. Symbols in ACL2 are actually very complicated objects but in this course we treat symbols as simple atomic objects.
- conses: A *cons* is an object containing two other objects. The name `cons` in Lisp is the name of the function that *cons*tructs a new object from its two constituent parts, which are called the *first* and *rest* (sometimes informaly called the "left" and "right", "head" and "tail", or "car" and "cdr", respectively). Conses are written with parentheses. We'll discuss them at length below.

Any Lisp object that is not a cons is said to be an *atom*. Thus, `23` is an atom. So are the string `"Hi there"` and the symbols `t`, `Monday`, and `A`. But `(A)`, `(A 1)`, and `((A 1) C D)` are not atoms. They are all conses. The rules for writing conses are discussed below. Just remember that if you see an object that starts with an open parenthesis – other than `nil` when you choose to write it as `()` – it is a cons. Any object that doesn't start with an open parenthesis is at atom.

Objects of different types are different. Thus, the strings `"HELLO"` and the symbol `HELLO` and the cons `(HELLO)` are different. There is a function, `equal`, that takes two arbitrary objects as input and returns `t` if they are the same object and `nil` otherwise.

It is possible for ACL2 programs to determine the type of each the object. You can think of each object having a hidden tag field that contains the type. For each of the five types, *typ*, there is a recognizer function that takes an object of any type as input and returns `t` if the object is of type *typ* and `nil` otherwise. We discuss the primitive recognizers when we present the primitive functions.

2.10.1 Lists

Conses are constructed by the two argument function `cons`. You can "cons together" any two objects, x and y, and get an object, say, c. The object c is said to be a *cons*. If you apply the function `first` to it you get x back, and if you apply the function `rest` you get y back.[4] The function `consp` is the recognizer for conses; it returns `t` if its argument was constructed by `cons` and `nil` otherwise.

We will use conses and `nil` to represent lists. For example, the list of length 3 containing `1`, `2`, and `3` is built this way:

```
(cons 1 (cons 2 (cons 3 nil)))
```

and prints this way: `(1 2 3)`.

Aside: ACL2 terms and ACL2 data objects are both written with parentheses but they are different sorts of things. A term is something containing variables, constants, and function calls and is meant to be evaluated. Data objects, on the other hand, are the *values* of terms, including the values of variables in the environment. We will discuss it at length later, but the list constructed by `(cons 'first (cons 'x nil))` prints as `(first x)`. If I write "`(first x)`" you can't tell *a priori* whether I mean the term that is the application of the function symbol `first` to the variable symbol `x`, or whether I mean the cons data object whose first element is the symbol `first` and whose second element is the symbol `x`. You have to decide which I mean by context. For example, if I say "the value of `(first x)` is ..." I am clearly using "`(first x)`" as a term, because I'm evaluating it. On the other hand, if I say "... evaluates to `(first x)`," the "`(first x)`" denotes a list data object because I'm using it as a value. If I say "... evaluates to the same thing that `(first x)` does", I'm using it as a term. I believe you will always be able to tell whether I'm referring to a term or a value. Just be aware that we're dealing with two sorts of entities – terms and objects – but using the same notation to denote them. The confusion could be totally avoided if Lisp adopted the idea of denoting list data objects with, say, square brackets, e.g., "`(cons 1 (cons 2 (cons 3 nil)))` $\Longrightarrow$ `[1 2 3]`." But it doesn't and there is a *very* good reason, although we will not explore it in this course: in Lisp, terms *are* objects. This allows Lisp programs to construct and run other Lisp programs. Lisp is its own metalanguage. This is one of the most powerful features of Lisp. But we are not going to exploit this fact in this course. *End of Aside*

[4] It if helps, you can think of `cons` as a constructor for a new Object of class `Cons` with two fields, `first` and `rest`. However, in ACL2 there is no way to modify the contents of the fields.

Returning to the representation of lists by conses, suppose we wanted to build a list of the squares of `1`, `2`, and `3`. Then we could write:

```
(cons (* 1 1) (cons (* 2 2) (cons (* 3 3) nil)))
```

and construct the list `(1 4 9)`.

More generally, suppose we have $n > 0$ expressions, α_1, α_2, ..., α_n and that the value of α_i is the object c_i. Then

```
(cons α1
      (cons α2
            ...
            (cons αn
                  nil)...))
```

constructs the list of the values and prints like this:

(c_1 c_2 ... c_n).

The list above has n elements; n is called the *length* of the list. The first element is c_1, the second c_2, etc. The last element, the nth, is c_n. The `nil`, supplied as the "very last" argument in the `cons`-expression above, is *not* an element of the list.

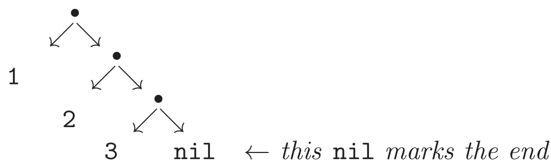
I'll say this several more times because students always have trouble with it!

To construct a list of n elements you need n conses, nested to the right as shown. This means you have to supply a total of $n + 1$ things to those conses, the α_i and that final `nil`.

You can think of `nil` as a representative of the empty list and you can think of `(cons` x y`)` as *adding a new first element*, namely, the value of x, to the list represented by the value of y.

So to build `(1 2 3)`, for example, you first build the "singleton" list containing just `(3)` by consing `3` onto the empty list `nil`. To do this, you could evaluate `(cons 3 nil)`. Then you use another cons to add the element `2` to that, and then you use another cons to add the element `1` onto that. The `cons`-expression is thus `(cons 1 (cons 2 (cons 3 nil)))`.

One way to think of a cons object is that it is a binary tree. For example, `(1 2 3)` might be drawn as shown below. Each big dot (•) in this picture is a cons that "points to" one element (in its first component) and "points to" the rest of the elements (in its rest component):



The `nil` is a *terminal marker* that means "there are no more elements." It is not an element of the list represented by this structure!

The *spine* of the list is that chain of conses. (There would be other conses elsewhere if the elements of our list had themselves been lists!) That spine is sometimes called the *rest chain* because it is just the successive rests of the list until you run out of conses. The number of conses in the spine is the length of the list because there is one element for every cons in the spine.

The names “first” and “rest” come from the use of conses to represent non-empty lists.[5]

So how do you get the second element of a list? The second element of x is the first element of the rest of x.

```
(defun second (x) (first (rest x)))
```

Suppose `x` is bound in the environment to the list object `(1 2 3)`. Then

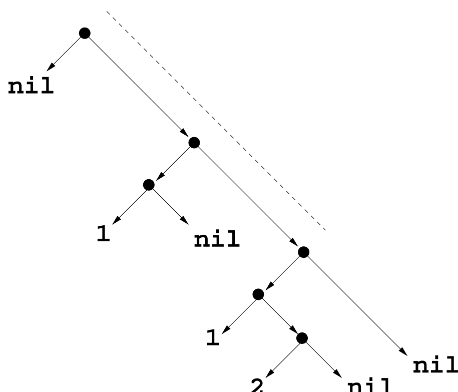
```
(first x) ⟹ 1
(rest x)  ⟹ (2 3)
(second x) = (first (rest x)) ⟹ 2
```

Here are some example printed lists and an English description of them.

`()`	the list of length 0, containing no elements, aka `nil`; you may write `nil` this way but ACL2 will never print it this way
`(1)`	the list of length 1, containing the element `1`
`(1 2)`	the list of length 2, containing the element `1` followed by the element `2`
`(() (1) (1 2))`	the list of length 3, containing the three previously shown lists as elements, in order; you will never see ACL2 print this list this way; see the next example
`(nil (1) (1 2))`	same as the list immediately above, printed as ACL2 will print it

If I were to draw the last list above as a binary tree, here is what I would draw:

[5] In Lisp the “primitives” are named `car` and `cdr`. The functions `first` and `rest` are defined, e.g., `(defun first (x) (car x))`. You might see ACL2 talk about `car` or `cdr` for that reason.



The dashed line is drawn along the spine, which contains three conses. Hanging off the spine in the three "first" positions are the three elements, the first being `nil`, the second being `(1)`, and the third being `(1 2)`.

• **Question 47**: Write the `cons` expression to construct `(7 0)`. •

• **Question 48**: Write the `cons` expression to construct `((1) (2))`. •

I'll answer this question to show you how I work out such questions. The list we're creating has two elements, i.e., it is of the form (α β). So I know the answer will look like this:

```
(cons α
      (cons β
            nil))
```

where α is the `cons`-expression to create `(1)` and β is the `cons`-expression to create `(2)`. But each of these are singleton lists created by `(cons` δ `nil)`, where δ is `1` for α and `2` for β.

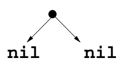
So the answer is:

```
(cons (cons 1 nil)
      (cons (cons 2 nil)
            nil))
```

I can test this by evaluating it in ACL2. I'll get `((1) (2))`.

• **QUESTION 49**: Write the `cons` expression to construct `(nil ((t nil)) t)`. Test your answer with ACL2. •

Many students are confused about whether the empty list, `nil`, is the same thing as the list containing the empty list, `(nil)`, or `(())`. `Nil` and `(nil)` are different! The first has no conses in its spine. The second has one. The first has length `0` and the second has length `1`. One would be "drawn" as `nil` and the other is drawn as



I think the confusion that `nil` and `(nil)` are the same comes from the idea that "`nil` is nothing", so "the list containing `nil`" is misunderstood as "as the list containing nothing." But `(nil)` is the list that contains the list containing nothing, so it contains something.

2.10.2 Dot Notation

What happens when we cons a new element, say 3, to something that doesn't naturally represent a list, like 45? What do we get?

We get a cons, whose first is `3` and whose rest is `45`. But how do we print it?

Many students want to say that `(cons 3 45)` should print as `(3 45)` but that is wrong! The length of the object constructed by `(cons 3 45)` is `1`, because there is one cons in its spine. But the length of `(3 45)` is `2`. So they can't be the same.

In this section, I tell you how to print and read such "weird" objects as the one constructed by `(cons 3 45)`. But I will never use them in this course and I will never expect you to use them!

Nevertheless, you *will* encounter them because in some of the homeworks you'll make mistakes (yes!) and do things like cons `nil` onto `3` instead of `3` onto `nil`. ACL2 won't protest because there is a perfectly good cons object with first `nil` and rest `3`. But you should at least be alert to how they appear when they print out because if you see such an object you know your homework answer is wrong! It's wrong because I won't ask you to construct such conses.

We can think of *any* object, c, as representing a sequence of objects. How? If c is a cons, then it represents the sequence whose first element is the `first` of c and whose remaining elements are those in the sequence represented by the `rest` of c. And if c is not a cons, then it is a terminal marker that means "there are no more elements," i.e., such a c represents the empty sequence. By applying this process to a cons c you should be able to recover a sequence of $n > 1$ elements, c_1, c_2, $\ldots$ c_n, and a terminal marker.

So here is how to print any Lisp object c.

- If c is not a cons it is a number, symbol, string or something else you know how to print.
- If c is a cons, then let $c_1, c_2, \ldots, c_n$ be the n successive elements recovered from c by the process above and let d be the terminal marker.
  - If d is `nil`, print c the way I've already told you: `(`c_1 c_2 `...` c_n`)`.
  - If d is not-`nil`, print c like this: `(`c_1 c_2 `...` c_n `.` d`)`.

So `(cons 3 45)` prints as `(3 . 45)`.

`(cons 1 (cons 2 (cons 3 45)))` prints as `(1 2 3 . 45)`.

Notice that there is always a dot in this printing. If you ever see a dot in your homework output, you've answer my question incorrectly because I'll never want you to construct any list except one terminated by `nil`.

Lists terminated by `nil` are sometimes called *true lists* to distinguish them from lists like `(1 2 3 . 45)`. We'll also call them `nil`-*terminated* lists.

2.10.3 Using Lists to Represent other Structures

Lists can be used to represent many structures. Here is a list of symbols that might be used to name the days of the week.

```
(Sunday Monday Tuesday Wednesday Thursday Friday Saturday)
```

This same list could have been written

```
(sunday monday tuesday wednesday thursday friday saturday)
```

or

```
(SUNDAY MONDAY
 TUESDAY WEDNESDAY
 THURSDAY FRIDAY SATURDAY).
```

If we were writing an ACL2 program that took as input a symbol, `x`, and answered the question "Is `x` the name of day of the week?" we might use this constant in our code. If I were going to write such a function, I would first write the general-purpose function `mem` so that `(mem e lst)` answers the question "is `e` an element of the list `lst`?" Then my code for recognizing day names could be:

```
(defun day-namep (x)
  (mem x '(SUNDAY MONDAY
           TUESDAY WEDNESDAY
           THURSDAY FRIDAY SATURDAY)))
```

Note that I have to quote the list constant we've been talking about when I use it as an expression in my code. Essentially I'm telling Lisp that I'm using this as an object rather than as a term. Why? Because if if don't quote it:

```
(defun day-namep (x)
  (mem x (SUNDAY MONDAY
          TUESDAY WEDNESDAY
          THURSDAY FRIDAY SATURDAY)))
```

I would be asking whether `x` is an element of the list computed by the *function* `SUNDAY` applied to the six "variables" after it. More on this later.

We might use lists to store alternative equivalent names for the days, e.g.,

```
((SUNDAY    1 SUN SU)
 (MONDAY    2 MON M)
 (TUESDAY   3 TUE TU)
 (WEDNESDAY 4 WED W)
 (THURSDAY  5 THU TH THUR THURS)
 (FRIDAY    6 FRI F)
 (SATURDAY  7 SAT SA)).
```

Note that we added white space to our display of the list to emphasize that for each day name we give a number equivalent, a 3-letter equivalent, and then various short forms. Using this list constant, we might write a function `long-day-name` that takes an alleged day name and returns the long version of the name, e.g., both `(long-day-name 'W)` and `(long-day-name 4)` could return the object `WEDNESDAY`. To write such a function we'd need to search each element of our list, looking for the name we were given and then return the first element of the element in which we found it.

Note that the example `(long-day-name 'W)` illustrates the need for quoting again. We mean to call `long-day-name` on the symbol `W`. But if we wrote `(long-day-name W)` then we'd be calling `long-day-name` on the value of the variable `W`. More on this below.

What should `long-day-name` do if it is given an object that is not a day name? First, this is not quite the right question in ACL2, because `long-day-name` doesn't *do* anything! It just returns a value. So what value should it return if it is given an unacceptable argument? There are two common alternatives. One would be to "default" the argument to an acceptable one, e.g., `long-day-name` could implement the convention that any unacceptable day name is treated like it were `SUNDAY`. This would be quite surprising if you tried to make a plane reservation on Saturday, typed `S` to specify the day, and unknowingly got a reservation on Sunday because `(long-day-name 'S)` evaluated to `SUNDAY`.

A second common alternative would be for `long-day-name` to return `nil` if it is given an unexpected input. The caller would then have to handle this case by asking whether `long-day-name` returned `nil` or something else. This would be coded something like this:

```
(if (long-day-name day)
    (...make a reservation...)
    "You provided an illegal day name!")
```

This illustrates Lisp's peculiar habit of using functions as predicates. That is, `(long-day-name day)` can be treated as a predicate that returns true or false, and, when true, its value is the long name of the given day. This works provided none of the days is named `nil`!

Here are some other examples of how an ACL2 programmer might use lists.

A table listing names and how many people in a class have a given name might be represented by this list:

```
(("Emma"        4)
```

```
("Jacob"       2)
("Isabella"    2)
("Emily"       1)
("Michael"     1)
("Ethan"       0))
```

Given such a table, one might contemplate how to answer such questions as “How many `"Michael"`s are in the class?” or “How many students are in the class?”; one might also contemplate how to build a new table when a new student enters the class or two classes are combined.

A normalized polynomial in the variable symbol “x”, such as $6x^2 + 5x - 21$ might be represented as `((6 2) (5 1) (-21 0))`. One might contemplate how to factor a normalized polynomial in this form, to create, for example, the two factors, `((3 1) (7 0))` and `((2 1) (-3 0))`, i.e., $3x + 7$ and $2x - 3$.

An arbitrary algebraic expression, like $(3x+7) \times (2x-3)$, might be represented by `(* (+ (* 3 x) 7) (+ (* 2 x) -3))`, and one might contemplate how to put it into some cannonical form.

A Java method, such as

```
public static int ifact(int n){
    int a = 1;
    while (n>0) {a = n*a; n = n-1;}
    return a;
}
```

might be represented as

```
(public static int ifact ((int n))
  (assign (int a) 1)
  (while (> n 0) (assign a (* n a)) (assign n (- n 1)))
  (return a))
```

and one might contemplate how to compile it (generate JVM byte code for it) or how to answer such questions as whether it might divide by `0`.

An English sentence like “The boy went home.” might be represented as `(the boy went home)` and one might contemplate parsing it into a form like

```
(SENTENCE (NOUN-PHRASE (DETERMINER the) (NOUN boy))
          (VERB-PHRASE (VERB went) (NOUN home)))
```

or even answering questions, like `(where did the boy go ?)`.

2.10.4 When Do You Use Quote?

Recall the "Aside" on page 35 about the possible confusion of objects with terms.

If you want to use an object as term denoting that object, you must quote the object unless it is a number, a Boolean (`t` or `nil`), or a string.

Putting a single quote mark in front of any ACL2 object creates an ACL2 term that evaluates to that object. You may quote numbers, Booleans, and strings if you want, but there is no need to because those objects always evaluate to themselves anyway.

Here are some examples just to drive home the point:

ACL2 expression	*value*
`W`	value of the variable `W` in the environment
`'W`	the symbol object `W`
`(Mon Wed Fri)`	value of function `Mon` applied to the values of the variables `Wed` and `Fri` in the environment
`'(Mon Wed Fri)`	the list of length 3, `(Mon Wed Fri)`, containing the symbols `Mon`, `Wed`, and `Fri`; this object could be constructed by evaluating `(cons 'Mon (cons 'Wed (cons 'Fri nil)))`
`t`	the symbol `t` (a true object)
`'t`	the symbol `t` (a true object)
`nil`	the symbol `nil` (the false object)
`'nil`	the symbol `nil` (the false object)
`23`	the integer `23`
`'23`	the integer `23`

What is the result of evaluating `(rest '(Mon Wed Fri))`? It is the object `(Wed Fri)`.

Does the equation

`(rest '(Mon Wed Fri))` = `(Wed Fri)`

express this idea? No! This is not an equation because an equation is supposed to have *expressions* on both sides! This thing has an expression on one side and a value (object) on the other. If I read it as an equation I'd be saying that applying `rest` to the value of

`'(Mon Wed Fri)`, produces the same thing as applying the function `Wed` to the value of the variable `Fri`. Not likely.

When we wish to say "expression α evaluates to the object β" we will write

$\alpha \Longrightarrow \beta$.

If I write that, then this is a true equation:

$\alpha =$ 'β.

This distinction doesn't arise before we have list- and symbol-valued data. For example

```
(+ 2 2) ⟹ 4
(+ 2 2) = '4
(+ 2 2) = 4
```

are all accurate statements.

If you are confused what a function does on a quoted list constant it is sometimes easier to replace the quoted constant by the `cons`-expression that produces it.

```
'(Mon Wed Fri) = (cons 'Mon '(Wed Fri))
               = (cons 'Mon (cons 'Wed '(Fri)))
               = (cons 'Mon (cons 'Wed (cons 'Fri '())))
               = (cons 'Mon (cons 'Wed (cons 'Fri nil)))
```

We can compute with such equations. For example, here are three important equations about `consp` and `rest`:

```
(consp nil)        = nil
(consp (cons u v)) = t
(rest  (cons u v)) = v
```

These hold *no matter what expressions* are put in for u and v.

So by equational reasoning:

```
(consp '(Mon Wed Fri)) = (consp (cons 'Mon '(Wed Fri)))
                       = t
(rest  '(Mon Wed Fri)) = (rest (cons 'Mon '(Wed Fri)))
                       = '(Wed Fri)
                       = (cons 'Wed (cons 'Fri nil))


(consp '(Wed Fri))     = (consp (cons 'Wed '(Fri)))
                       = t
(rest  '(Wed Fri))     = (rest (cons 'Wed '(Fri)))
                       = '(Fri)
```

```
                        = (cons 'Fri nil)


(consp '(Fri))          = (consp (cons 'Fri nil))
                        = t
(rest  '(Fri))          = (rest (cons 'Fri nil))
                        = nil


(consp nil)             = nil
```

Also recall the discussion of the "plug-in-the-actuals" evaluation scheme on page 15.

The official way to evaluate (*f* a_1 ... a_n), where *f* is a defined function, is to evaluate the a_i to obtain the actuals, bind the formals of *f* to the actuals to get a new environment, and evaluate the body of *f* in that new environment.

But the plug-in-the-actuals scheme suggested we can just substitute the actual values for the formals in the body and evaluate that. I said that was subtly flawed. Here I explain why, and what to do about it.

Let `foo` be defined

```
(defun foo (x)
  (if x t nil)).
```

Let's evaluate `(foo (cons 'zp (cons 23 nil)))` under the two schemes.

Official Scheme: Evaluate the actual expression to obtain the list object `(zp 23)`. Bind the formal `x` to this object to get a new environment *env′*. Evaluate `(if x t nil)` in this environment. To evaluate the `if`, evaluate the test `x`. Since it is a variable, we look up its value in *env′* and get the list `(zp 23)`. Since the list `(zp 23)` is not `nil`, the test is true and we evaluate and return `t`. The official (correct) answer is the object `t`.

Bad plug-in-the-actuals: Evaluate the actual expression to obtain the list object `(zp 23)`. Substitute the actual for the formal, `x`, in the body of `foo` to get the instantiated body:

```
(if (zp 23) t nil).
```

To evaluate the instantiated body, evaluate the `if`, by evaluating the test `(zp 23)`. Since `23` is a non-0 natural number, `(zp 23)` evaluates to `nil`. Thus the `if` takes the false branch and evaluates and returns `nil`.

Note that the plug-in-the-actuals scheme produced the wrong answer. Why? Because when we plugged in `(zp 23)` for `x` we started treating the data object `(zp 23)` as a piece of program text and evaluated it.

The correct statement of the plug-in-the-values scheme is: evaluate the actual expressions to obtain the actuals, substitute the *quoted* actuals for the formals in the body to obtain the instantiated body, and evaluate that. *Note that by quoting the actuals we turn them from*

data objects to expressions that produce those data objects.

The instantiated body in the `(foo (cons 'zp (cons 23 nil)))` example above should have been:

```
(if '(zp 23) t nil).
```

The value of this expression is `t`, as it should be.

Finally, there is an even simpler "plug-in" evaluation scheme we could use: To evaluate (f a_1 ... a_n) for a defined function f, plug in the actual expressions for the formals in the body of f and evaluate the instantiated body.

That is, we could evaluate `(foo (cons 'zp (cons 23 nil)))` by producing the instantiated body

```
(if (cons 'zp (cons 23 nil)) t nil)
```

and then evaluating that. Note that this produces the correct answer.

The official scheme I have adopted is called *call-by-value* because we evaluate the actuals and then run the body on the actual values. The scheme just described is called *call-by-name* because we don't evaluate the actual expressions but just literally substitute them into the body and evaluate them when we need to. In a language like ACL2, call-by-value and call-by-name are equivalent.

Sometimes one method is more efficient than the other. Consider `(bar (baz 23))` where `bar` is defined as shown below. Suppose that `(baz 23)` is very expensive to evaluate (e.g., takes a long time).

```
(defun bar (x) (cons x (cons x nil)))
```

Let's compare the cost of a call-by-value evaluation of `(bar (baz 23))` and a call-by-name evaluation.

Call-by-value evaluates `(baz 23)` first, binds `x` to that value, and then evaluates `(cons x (cons x nil))`. But call-by-name substitutes `(baz 23)` for `x` and evaluates `(cons (baz 23) (cons (baz 23) nil))`. Thus, it will evaluate the super-expensive `(baz 23)` twice. In this case, call-by-value is much more efficient than call-by-name.

On the other hand, if `bar` were defined to be

```
(defun bar (x) 45)
```

then call-by-value needlessly evaluates `(baz 23)` while call-by-name doesn't evaluate it at all. In this case, call-by-value is much less efficient than call-by-name.

Call-by-name is an example of what is called *lazy evaluation* because it delays evaluation until the results are needed.

No matter what scheme you use with ACL2, if the scheme terminates and gives a value, the answer is the same as the "official" scheme.

2.10.5 Practice Recognizing ACL2 Objects

• **QUESTION 50**: For each of the displays in the first column below, classify it according to the type of object denoted. You cannot do this assignment in ACL2. Just edit your session transcript, after you've saved it, to insert a list of numbers from 1 to 21 and then write "num", "char", "str", "sym", "cons" or "none" according to the type of the "object" displayed. If the display does not denote an ACL2 object, write "none". These notes are incomplete. You may have to guess on some of these! Here are a few clues: ACL2 does not support numbers with decimal points; rational numbers may be written as fractions, e.g., `3/5`. Commas and dots are never used in the ACL2 objects in this book, except possibly in strings such as `"Hello, world."` An alleged "object" can fail be an object because it is really two or more objects. For example `123 45` is not an object, it is two objects. Use your common sense and let's see what happens.

	object	*num* A	*char* B	*str* C	*sym* D	*cons* E	*none*
1	`346`						
2	`2.167`						
3	`Hello`						
4	`Mary Ann`						
5	`(Hello Mary Ann)`						
6	`((A 1) (B 2))`						
7	`"Error: Bad PC!"`						
8	π						
9	`360.`						
10	`((A 1) B C)`						
11	`(A () (B C))`						
12	`(A))(B C)`						
13	`((A B C)`						
14	`UNDEF`						
15	`-1000`						
16	`(A B C` π`)`						
17	`(NIL)`						
18	`NIL`						
19	`()`						
20	`00023`						
21	`10,000`						

•

2.11 Primitive Functions (Assignment 4: 2 days)

Now that you know how to write down all the constants we will use, all that remains is to list the primitive function symbols you can apply to them. We use a small subset of ACL2 in this booklet.

`(if` x y z`)` — if x is true, then y, else z

`(equal` x_1 x_2`)` — `t` or `nil` according to whether x_1 and x_2 are the same object.

`(cons` x_1 x_2`)` — cons constructor; the value is the cons whose first component is x_1 and whose rest component is x_2. In this course, we'll follow the convention of insuring that x_2 is a `nil`-terminated list.

`(consp` x`)` — Boolean recognizer for conses.

`(first` x`)` — first element x if x is a cons; `nil` if x is not a cons.

`(rest` x`)` — rest component of x if x is a cons; `nil` if x is not a cons.

`(symbolp` x`)` — Boolean recognizer for symbols.

`(stringp` x`)` — Boolean recognizer for strings.

`(integerp` x`)` — Boolean recognizer for integers.

`(natp` x`)` — Boolean recognizer for non-negative integers.

`(+` x_1 x_2`)` — sum: $x_1 + x_2$; non-numeric arguments are treated as 0.

`(-` x_1 x_2`)` — difference: $x_1 - x_2$; non-numeric arguments are treated as 0.

`(*` x_1 x_2`)` — product: $x_1 \times x_2$; non-numeric arguments are treated as 0.

`(/` x_1 x_2`)` — quotient: x_1/x_2; if x_1 is non-numeric, it is treated as 0; if x_2 is non-numeric or 0 the value of (/ x_1 x_2) is 0.

`(<` x_1 x_2`)` — less than: `t` or `nil` according to $x_1 < x_2$; non-numeric arguments are treated as 0.

The talk above about non-numeric arguments highlights the fact that ACL2 is an *untyped language*. What does that mean? It means you can apply any function to any type of object. You can write weird expressions in ACL2 – like `(+ T 3)` and `(< -2 'MONDAY)` – and they have meaning. In particular, they have well-defined values[6]. In ACL2, `(+ T 3)` is equal to

[6] By “well-defined” I don't mean the values necessarily “make sense intuitively” but just that they are

`3` and `(< -2 'MONDAY)` is `T`.

In a *typed language*, like Java, such expressions are just ill-formed. They don't mean anything and never get evaluated because you're not allowed to write them.

If you read the talk about non-numeric arguments above you can figure out that `(+ T 3)` evaluates to `3` because the non-numeric argument `T` is treated as though it were `0`. So `(+ T 3)` = `(+ 0 3)` = `3`.

• **Question 51**: Explain the value of `(< -2 'MONDAY)`. •

Even though we will mainly use the natural numbers in this booklet, you should be aware that the arithmetic functions operate on all numbers. They are not limited to naturals or "closed" on the naturals. That is, giving them natural numbers for arguments does not guarantee their results are naturals. For example `(- 5 7)` is `-2` and `(/ 15 6)` is the rational `5/2`, which is a number between `2` and `3`.

2.12 Some Simple Definitions (Assignment 4 cont'd)

Here are some functions we'll use often.

2.12.1 Defined Propositional Functions

```
(defun and (p q) (if p q nil))

(defun or (p q) (if p p q))

(defun not (p) (if p nil t))

(defun implies (p q) (if p (if q t nil) t))

(defun iff (p q) (and (implies p q) (implies q p)))

(defun booleanp (x)
  (or (equal x t)
      (equal x nil)))
```

The first five are called the *propositional functions*, *propositional connectors*, or *logical operators* because they're used to form "propositions" like "x is a natural and less than y." Like all ACL2 functions, they can take any arguments you give them, but they're meant to be applied to Booleans.

A *propositional formula* is an ACL2 formula in which all the function symbols are propositional (`and`, `or`, `not`, `implies`, and `iff`) and the only constants are `t` and `nil`. Thus,

completely specified. In some languages, inappropriate arguments lead to unpredictable answers or behavior. That doesn't happen in ACL2.

`(implies (and a b) (not c))` is a propositional formula but `(implies (and a (natp i)) b)` is not because it mentions the function `natp`.

The last function above, `booleanp` recognizes the two Boolean values, returning `t` if its argument is a Boolean and `nil` otherwise. It is not considered a propositional function but it is handy to define here.

`(And p q)` is the *conjunction* of `p` and `q`. It is true if both `p` and `q` are true and it is false otherwise. In `(and p q)`, `p` and `q` are called the *conjuncts*.

Think in terms of people making statements about what they think is true. Suppose I say "*p* and *q*" and that I'm telling the truth. Then you know that *p* is true and you know that *q* is true. Similarly, if either *p* or *q* is false, then saying "*p* and *q*" is false. The formalization of "*p* and *q*" is `(and p q)`.

What does it mean if I say "`1` and `3`" as though it were a proposition that was either true or false? In normal usage, that would be nonsense. But its formalization in ACL2 as "`(and 1 3)`" has meaning because ACL2 is untyped: functions in ACL2 can be applied to anything, so "`(and 1 3)`" means *something*. You might think `(and 1 3)` is true because both `1` and `3` are true objects. You would be right, provided you're thinking of "true" as an adjective. `(And 1 3)` evaluates to a non-`nil` object.

• **Question 52**: What is the value of `(and 1 3)`? •

It might be hard to work that out from the informal description of `(and p q)` as the conjunction of `p` and `q`. But it is trivial to work it out from the definition of `and`.

Given

```
(defun and (p q) (if p q nil))
```

we know

```
(and 1 3)
=                                   {definition}
(if 1 3 nil)
=                                   {since 1 is different from nil}
3
```

• **Question 53**: What is `(and nil 7)`? •

• **Question 54**: What is `(and 'Monday "Hi")`? •

When you are not sure what something means, go back to the definition!

The fact that `and` is not Boolean is very unconventional. ACL2 does it this way only because that is the ANSI standard for Lisp. We will not exploit the facts that Lisp's `and` and `or` do not return Booleans.

`(Or p q)` is the *disjunction* of `p` and `q`. It is true if either (or both) of them is true and false otherwise. In `(or p q)`, `p` and `q` are called the *disjuncts*.

`(Not p)` is the Boolean negation of `p`: `(not t)` is `nil` and `(not nil)` is `t`. This is also called the *negation* or *logical negation* of `p`.

`(Implies p q)` is called an *implication.* We call `p` the *hypothesis* or *antecedent* and we call `q` the *conclusion* or *consequent.*

`(Implies p q)` can be read "if `p` is true, then `q` is true." So when is `(implies p q)` true? The *best* way to answer that question is to look at the definition:

```
(defun implies (p q) (if p (if q t nil) t))
```

According to this definition:

1. `(implies t t)` $\Longrightarrow$ `t`
2. `(implies t nil)` $\Longrightarrow$ `nil`
3. `(implies nil t)` $\Longrightarrow$ `t`
4. `(implies nil nil)` $\Longrightarrow$ `t`

We don't have to consider other possibilities, such as when `p` is `23` and `q` is the string `"Hello, world!"` because `p` and `q` are just tested by `implies`. All that matters about their values is whether they are `nil` or non-`nil`.

Now let's compare each of the four evaluations to the alleged English meaning of `(implies p q)`: "if `p` is true, then `q` is true." When is such a sentence true?

1. `(implies t t)` means "if `t` is true, then `t` is true." Surely that is a true statement, so the evaluation to `t` makes sense.

2. `(implies t nil)` means "if `t` is true, then `nil` is true." Surely that is a false statement, because `t` *is* true but the conclusion, "`nil` is true," is false. This is like me saying "If $2 + 2 = 4$, then $1 = 3$." It's just not so! Thus, the evaluation to `nil` makes sense.

3. `(implies nil t)` means "if `nil` is true, then `t` is true." Clearly, `nil` is not true. But it doesn't matter, because all I conclude is that `t` is true, which it is. So the sentence is true. For example, have I lied if I say "if $1 = 3$, then $2+2 = 4$?" No! The evaluation to `t` makes sense. In fact, "if ..., then `t` is true" is a true sentence, no matter what "..." is. So `(implies` α `t)` is `t`, no matter what α is.

4. `(implies nil nil)` means "if `nil` is true, then `nil` is true." Students often have trouble with this one. But read it literally. If I said "if `nil` is true, then `nil` is true" would I be lying? No! I wouldn't be saying anything very interesting or informative, but I'm telling the truth. There is an old English proverb you might have heard that uses this same kind of reasoning: "If wishes were horses, beggers would ride." It's a true saying *because* the hypothesis is false. The truth of `(implies nil` α`)` doesn't depend on α. No matter what you fill in for α, `(implies nil` α`)` is true because the hypothesis is false. So the evaluation to `t` makes sense.

A valuable lesson is that if `(implies p q)` is true, it *doesn't mean* `q` is true! Remember that! It just means `q` is true *if* `p` *is true* – and it might be true anyway!

The *converse* of `(implies` p q`)` is `(implies` q p`)`.

The *contrapositive* of `(implies` p q`)` is `(implies (not q) (not p))`.

Finally `(iff p q)` means the truth values of `p` and `q` are the same. Either they're both true or they're both false. So `(iff t t)` and `(iff nil nil)` are both true and `(iff t nil)` and `(iff nil t)` are both false. The name `iff` stands for "if and only if." There are many English phrasings of this. We might say "`p` is true if and only if `q` is true," or "`p` and `q` are propositionally equivalent," or "`p` and `q` are propositionally indistinguishable."

• **QUESTION 55**: The *exclusive or* of two arguments is true if exactly one of the two arguments is true, and false otherwise. It is defined in ACL2 as

```
(defun xor (p q)
  (if p
      (if q nil t)
      (if q t nil))).
```

Define an equivalent function, `my-xor`, whose body only calls the functions `and`, `or`, and `not`. Evaluate these four calls of your function in ACL2.

```
(my-xor nil nil) ⟹ nil
(my-xor nil t)   ⟹ t
(my-xor t nil)   ⟹ t
(my-xor t t)     ⟹ nil •
```

• **Question 56**: Write a term that checks whether exactly one of `p`, `q`, and `r` is true. By *checks* I mean the term returns true if the condition is true and false otherwise. Do not use `if`. •

• **QUESTION 57**: We say `x` is a *singleton* if `x` is a cons whose `rest` is `nil`. Define `singletonp` to recognize singletons. Do not use `if`. Run your function on these examples in ACL2.

```
(singletonp '()) ⟹ nil
(singletonp '(Mon)) ⟹ t
(singletonp '((Hi there))) ⟹ t
(singletonp '(Hi there)) ⟹ nil
(singletonp 23) ⟹ nil  •
```

• **Question 58**: We say `x` is a *doublet* if it is of the form `(`α β`)`. Define the recognizer for doublets, `doubletp`. Do not use `if`. Run your function on these examples.

```
(doubletp '()) ⟹ nil
(doubletp '(Mon)) ⟹ nil
(doubletp '((Hi there))) ⟹ nil
(doubletp '(Hi there)) ⟹ t
(doubletp '(Mon Tue)) ⟹ t
```

(doubletp 23) ⟹ nil •

• **Question 59**: We say `x` is *text-likep* if `x` is a symbol or a string. However, we don't consider the symbols `t` and `nil` text-likep. Define the recognizer, `text-likep`, for text-likep objects. Do not use `if`. Run your function on these examples.

```
(text-likep "Hi") ⟹ t
(text-likep 'temp) ⟹ t
(text-likep t) ⟹ nil
(text-likep nil) ⟹ nil •
```

• **QUESTION 60**: If something is a cons, then it is equal to the result of consing its own `first` and `rest` together. This statement is always true. (a) Define the function `test-q60` to test this statement for a given object, i.e., to return `t` or `nil` to indicate whether the statement is true for that particular object. Do not use `if` in your definition. (b) Test your definition on the following examples.

```
(test-q60 '(1 2 3)) ⟹ t
(test-q60 '(Mon)) ⟹ t
(test-q60 '((Mon) (Tue) (Wed)))  ⟹ t
(test-q60 23) ⟹ t
```

(c) Is it possible for someone who cannot see how you defined `test-q60` to determine whether it tests the statement made at the beginning of this question? •

• **Question 61**: If `x` is not a cons then its `first` is `nil`. This statement is always true. Define the function `test-q61` to test it for a given object. Do not use `if` in your definition. Test your definition on the following examples.

```
(test-q61 23) ⟹ t
(test-q61 nil) ⟹ t
(test-q61 '(Mon)) ⟹ t
(test-q61 '(Mon Tue)) ⟹ t •
```

• **Question 62**: `X` is equal to `0` or `1`. This statement is not always true, but it can still be said. Define `test-q62` to test this statement for a given object. Do not use `if` in your definition. Then exhibit an example call that makes it true and an example call that makes it false.

(test-q62 α) ⟹ t
(test-q62 β) ⟹ nil •

• **Question 63**: Exactly one of the following is true: `x` is a cons, a symbol, or a string. This statement is not always true but it can still be said. Define `test-q63` to test this statement for a given object. Do not use `if` in your definition. Show an example that makes it true and one that makes it false.

(test-q63 α) $\Longrightarrow$ t
(test-q63 β) $\Longrightarrow$ nil •

• **Question 64**: If you have three naturals and they are all different, then adding them up never produces 0. Say that, without using if. •

• **QUESTION 65**: If you have three naturals and they are all different, then one is at least the sum of the other two. This statement is not always true. Define the function test-q65 to test this statement; do not use if in the definition of test-q65. Show an example that makes it true and an example that makes it false, using ACL2.

(test-q65 α_1 α_2 α_3) $\Longrightarrow$ t
(test-q65 β_1 β_2 β_3) $\Longrightarrow$ nil •

• **Question 66**: If x is a cons whose first is y and z is a string, then if the first of x is z, then y is a string. (a) Define test-q66 to take three arguments and test this statement. Do not use if in the body of your definition. (b) Does test-q66 always return t? If not, show an evaluation of (test-q66 α β γ) that is false. •

• **Question 67**: Conses and strings are disjoint. A way to say this is that if x is a cons then it isn't a string, and vice versa. Say that, without using if. •

• **Question 68**: If one of y and z is a cons and the other is a natural, and if x is either y or z then x is not text-like. Say that, without using if. You can use the predicate text-likep from Question 59 (page 53). •

• **Question 69**: The product of the sum and difference of x and y is positive, provided x and y are naturals. Say that without using if. •

People are really good at inventing new ways to say the same old things. I guess we get bored with repetition of patterns of speech. So we might say "if p then q," but we might also way "q, provided p," "q if p," "when p then q," and even things like "q or else p is false." Get used to it!

• **Question 70**: Let the *parity* of a natural be the symbol even or odd with the obvious meaning. Suppose the function parity returns the parity of its argument. You don't have to define parity, just pretend you have. The parity of the sum of two naturals is even if they have the same parity. Say that, without using if. •

• **Question 71**: If x and y and their product all have the same parity then all three are even. Say that, without using if. See the previous question for an explanation of the function parity. •

There are a couple of natural ways to interpret the last question. People tend to drop words and assume the reader will fill them in. Did the speaker mean all three are literally even or did the speaker mean the parities of all three are even? There's no way to know except to ask. Get used to trying to say precisely what somebody else has said vaguely or ambiguously!

2.12.2 Defined List Functions

```
(defun endp (x) (not (consp x)))
```

Note that `(endp x)` is just another way to write that `x` is not a cons. We typically use `endp` to recognize when we've reached the terminal marker of a list.

• **Question 72**: We say a *triple* is a list of the form $(x\ y\ z)$. We call the three components of a triple `trip1`, `trip2`, and `trip3`. Define `triplep` to recogize triples. Additionally, define the three accessor functions so that they return `nil` on non-triples and otherwise return the respective components of the triple. Here are some examples:

```
(triplep '(1 2 3)) ⟹ t
(triplep '((A B) 2 (Mon Wed Fri))) ⟹ t
(triplep '(1 2 3 4)) ⟹ nil
(triplep '(1 2)) ⟹ nil
(triplep '(1 2 3 .  45)) ⟹ nil
(triplep (cons 1 (cons 2 (cons 3 45)))) ⟹ nil
(trip1 '(11 22 33))  ⟹ 11
(trip2 '(11 22 33))  ⟹ 22
(trip3 '(11 22 33))  ⟹ 33 •
```

• **Question 73**: Define `set-trip1` so that `(set-trip1 x y)` returns a triple (see the previous question) with the same components as `x` except for its `trip1` which is set to `y`. If `x` is not a triple, the `set-trip1` returns `x` as is. Thus, `(set-trip1 '(1 2 3) 55)` should return `(55 2 3)`. But `(set-trip1 '(1 2 3 4 5) 55)` returns `(1 2 3 4 5)`. •

• **Question 74**: Define `set-trip2` and `set-trip3` analogously to `set-trip1` in the previous question. •

• **Question 75**: Is the following formula true for all values `x`? If not, show a value for `x` that makes it false.

```
(implies (consp x) (not (endp x))) •
```

• **Question 76**: Is the following formula true for all values `x`? If not, show a value for `x` that makes it false.

```
(implies (consp x) (equal (endp x) nil)) •
```

• **Question 77**: Is the following formula true for all values `x`? If not, show a value for `x` that makes it false.

```
(implies (and (consp x) (endp x)) (equal x nil)) •
```

• **Question 78**: Is the following formula true for all values `x`? If not, show a value for `x` that makes it false.

```
(implies (endp x) (equal x nil)) •
```

• **Question 79**: Is the following formula true for all values `x`? If not, show a value for `x` that makes it false.

```
(implies (and (equal (first x) nil)
              (equal (rest x) nil))
         (endp x)) •
```

• **Question 80**: Is the following formula true for all values `x` and `y`? If not, show values for `x` and `y` that make it false.

```
(implies (and (consp x)
              (equal (first x) (first y))
              (equal (rest x) (rest y)))
         (and (consp y)
              (equal x y)))
```

• **Question 81**: Is the following formula true for all values `x` and `y`? If not, show values for `x` and `y` that make it false.

```
(implies (and (consp x)
              (consp y)
              (equal (first x) (first y))
              (equal (rest x) (rest y)))
         (equal x y))
```

2.12.3 Defined Arithmetic Functions

```
(defun <= (x y) (not (< y x)))
(defun > (x y) (< y x))
(defun >= (x y) (<= y x))
```

The functions above should be self-explanatory from your knowledge of algebra.

```
(defun zp (x)
  (or (not (natp x)) (equal x 0)))
```

This definition finally tells you everything there is to know about zp: it is `t` if x is not a natural number or is `0`, and nil otherwise.

For example, recall

```
(defun fact (n)
```

```
  (if (zp n)
      1
      (* n (fact (+ n -1)))))
```

We know `(fact 3)` is `(* 3 (* 2 (* 1 1)))` = `6`, as previously computed on page 16. But what is `(fact -3)`? If we had defined `fact` so that the test were `(equal n 0)` instead of `(zp n)`, then `(fact -3)` would run forever and try to compute `(* -3 (* -4 (* -5 (* -6 ...))))`. By terminating when `(zp n)` is reached instead of when `0` is reached, we make `(fact -3)` be 1. That is, `fact` treats all non-natural inputs just as it treats `0`.

If we recurse by testing `(zp n)` and decrement `n` when the test fails then our recursive function will always terminate.

A related and often useful function is

```
(defun nfix (x)
  (if (natp x)
      x
      0))
```

We think of `(nfix x)` as "casting" `x` to a natural number by returning `x` if `x` is a natural and returning `0` otherwise.

2.13 Defining New Functions (Assignment 4 cont'd)

You've seen the `defun` command. Here are a few brief tips.

To introduce a new function f of arity n, do

(defun f ($v_1 \dots v_n$)
  β)

where

1. f is a symbol that has never been defined before,
2. the v_i are variable symbols and no two are the same, and
3. β is an ACL2 term and
   (a) every function called by β has already been defined (except for f itself), and
   (b) there are no free variables in β except the v_i.
4. some natural number measure of the arguments gets smaller every time the function recurs.

If you try to define a function and ACL2 doesn't accept it, you've probably violated one of the rules above. The fourth condition insures that the function terminates when run on any object. If the system rejects a `defun` because it could not prove termination, switch to `:program` mode.

Remember: you can tell which mode ACL2 is in by the prompt. If the prompt is

```
ACL2 !>
```

then you are in logic mode and "guards" (i.e., type restrictions) are being checked and you should get into program mode by typing this to the prompt:

```
ACL2 !>:program
```

Then it will be in program mode but still checking guards; the prompt will look like this:

```
ACL2 p!>
```

To turn off guard checking (which I advise new users to do) do this:

```
ACL2 !>:set-guard-checking nil
```

The prompt will then look like this:

```
ACL2 p>
```

Here are some example *bad* definitions – ones that violate the rules.

```
(defun cons (x y) (+ x y))           ; Note 1
(defun f (x x) (+ x x))              ; Note 2
(defun even (n)                      ; Note 3(a)
  (if (zp n) t (odd (- n 1))))
(defun odd (n)
  (if (zp n) nil (even (- n 1))))
(defun f (x) y)                      ; Note 3(b)
(defun f (x) (not (f x)))            ; Note 4
```

It should be clear that "definitions" 1 and 2 are "bad." The first redefines an existing function (however, see the ACL2 documentation topic `u` (for undoing a definition) or `redef`). The second doesn't tell us how to evaluate `(f 1 2)`. The reason 3(a) is "bad" is that `even` calls `odd` before `odd` is defined. But since `odd` calls `even`, we can't introduce `odd` first either. This is an example of *mutual recursion*. We will avoid mutual recursion in this course. Definition 3(b) is "bad" because the value of `(f 3)` might be `5` in one execution and `7` in another, depending on the value of `y`. Finally, definition 4 is bad because nothing gets smaller when we recur. If we were to accept this definition, then we'd have `(f x)` = `(not (f x))` and if that is true then `t`= `nil`.

The restriction that some natural number measure of the arguments get smaller in every recursive call doesn't mean some argument gets smaller. Can you think of why this function

terminates?

```
(defun f (x y)
  (if (and (natp x)
           (natp y)
           (< x y))
      (f (+ 1 x) y)
      x))
```

It terminates because the absolute value of the difference of `x` and `y` decreases. Since the "absolute value of the difference" of two naturals is a natural, that's an acceptable "measure" of the size of the arguments. The ACL2 definitional principle is actually more powerful than described here but we won't go into it.

You may occasionally want to define a function that ignores one or more of its arguments. But for example, this definition

```
(defun foo (x) 23)
```

is rejected by ACL2. You must tell ACL2 that you are intentionally ignoring an argument.

```
(defun foo (x)
  (declare (ignore x))
  23)
```

With this definition in place, `(foo 3)` $\Longrightarrow$ `23` and `(foo 100)` $\Longrightarrow$ `23`; i.e., `foo` is a *constant function* of one argument that returns `23`.

2.14 Recursion on Lists (Assignment 5: 5 days)

In this section I ask you to define some recursive functions on lists. But I answer a few of the questions myself first. Study my answers before you answer the other questions!

The basic ideas behind all of the recursive functions on lists here are that (a) we'll use `endp` to recognize whether the object we're exploring is empty or non-empty, (b) if it is empty, then it is just the terminal marker and we solve the problem for the empty list, (c) if it is non-empty then it has a first element, accessed by `first`, and some remaining elements, accessed by `rest`. In this case, (c1) we call the function recursively on the `rest` (when necessary) and then (c2) process `first` (as necessary) to convert the smaller solution to the bigger one. Our functions terminate because the length of the list keeps getting smaller. Watch!

• **Question 82**: Define the function `len1` so that `(len1 x)` is the number of elements in the list `x`. (I.e., `(len1 x)` is the number of `cons` nodes in the spine of `x`.) For example,

```
(len1 nil) ⟹ 0
```

```
(len1 '(A B C)) ⟹ 3
(len1 '((A 1) (B 2) (C 3) (D 4))) ⟹ 4
(len1 '((A B C D E F G H I J K L))) ⟹ 1 •
```

My Thinking: Instead of just giving my answer, I'll explain my thinking. On this one problem I will include my "official" answer at the end so you can see what I expect you to turn in.

This function will be recursive. Is that obvious to you? For me, the intuition is "this function counts things and I don't know how many things there'll be, so I will have to just keep counting till I get to the end." Anytime you have a thought like "keep doing something till you get to the end" you're going to need recursion to describe it. If you're going to "keep doing something" to a list, then you must consider (at least) two cases: is the list empty or not?

So we can start by writing this:

```
(defun len1 (x)
  (if (endp x)
      __α__
      __β__))
```

Now remember the three creative things we have to do: (a) find a small version of the problem we can solve (the base case), (b) find a way to reduce a big problem to a smaller one, and (c) find a way to convert the smaller solution to the big solution. Here, α will be our solution to (a) and β will encode our solutions to (b) and (c). When dealing with lists, step (b) is almost always to apply the function to the rest of the list.

(a) what is the length of the empty list? Clearly, α should be `0`.

(b) If `x` is not empty, it has a `first` element and some other elements in `(rest x)`. What is a smaller version of the problem of finding the number of elements in `x`? Finding the number of elements in `(rest x)` is a smaller problem. We can do that with `(len1 (rest x))`: it is the number of elements in the rest of x.

(c) How can we convert the answer for the smaller problem into the answer for `x`? Put differently, how many more elements are there in `x` than in the rest of `x`, when `x` is non-empty? There is `1` more element in `x`, namely `(first x)`.

Here's another way to approach the problem. If `x` is non-empty, then it is of the form (δ_1 δ_2 ...), where δ_1 is `(first x)` and (δ_2 ...) is `(rest x)`. If you're new at this, you probably feel stymied because you can't look at all the elements, you can just look at the "current" one, δ_1, and have an arbitrarily long list of things to go (δ_2 ...). But now imagine a friend comes up and says, "Hey! I can tell you how many elements are in (δ_2 ...). The answer's a." Your response would probably be a huge sigh of relief, "Cool! I don't know how you did that, but it means the answer to my question is $a + 1$. Thanks!"

Now realize you *always* have that friend, ready to solve the smaller problem: it's you! You just have to realize that the friend is standing beside you ready to be asked! Furthermore, your friend can correctly answer the question for *any* smaller problem!

Putting (b) and (c) together, we see that β is `(+ 1 (len1 (rest x)))`.

The final answer is thus:

```
(defun len1 (x)
  (if (endp x)
      0
      (+ 1 (len1 (rest x)))))
```

We can test this with `(len1 '(Mon Wed Fri))` and get the answer `3`. If we try `(len1 nil)` we get 0. These evaluations can be expressed

```
(len '(Mon Wed Fri)) ⟹ 3
(len nil) ⟹ 0.
```

Since this is our first evaluation with lists, let's go slowly.

The following equations should be obvious, given the definition of `(endp x)` as `(not (consp x))` and the table of equations on page 44.

```
(endp '(Mon Wed Fri)) = nil
(rest '(Mon Wed Fri)) = '(Wed Fri)
(endp '(Wed Fri)) = nil
(rest '(Wed Fri)) = '(Fri)
(endp '(Fri)) = nil
(rest '(Fri)) = nil
(endp nil) = t
```

Given those equations and the definition of `len` we can compute

```
(len1 '(Mon Wed Fri))
=                                                  {def len1}
(if (endp '(Mon Wed Fri))
    0
    (+ 1 (len1 (rest '(Mon Wed Fri)))))
=                                  {endp and rest equations above}
(if nil
    0
    (+ 1 (len1 '(Wed Fri))))
=                                              {(if nil ...)}
(+ 1 (len1 '(Wed Fri)))
=                               {len1 and steps analogous to above}
(+ 1 (+ 1 (len1 '(Fri))))
=                               {len1 and steps analogous to above}
(+ 1 (+ 1 (+ 1 (len1 nil))))
=                                                      {len1}
(+ 1 (+ 1 (+ 1 (if (endp nil)
                   0
```

```
                   (+ 1 (len1 (rest nil)))))))
=                                                      {endp equation above}
(+ 1 (+ 1 (+ 1 (if t
                   0
                   (+ 1 (len1 (rest nil)))))))
=                                                             {(if t ...)}
(+ 1 (+ 1 (+ 1 0)))
=                                                            {arithmetic}
3
```

Since

```
(endp x) = (not (consp x))
```

we could use replacement of equals by equals to derive this definition of `len1`:

```
(defun len1 (x)
  (if (not (consp x))
      0
      (+ 1 (len1 (rest x)))))
```

Since

```
(if (not x) y z) = (if x z y)
```

we could use replacement of equals by equals to derive this

```
(defun len1 (x)
  (if (consp x)
      (+ 1 (len1 (rest x)))
      0))
```

All these definitions are equivalent, and we can use familiar replacement of equals by equals to derive one from the other. This is a trivial illustration of why logic is so important to computer science:

> *Programs are just mathematical statements and their properties can be derived using symbolic manipulation like you learned in algebra.*

The only problem is that the "mathematics" of Java and C++ and other commercial languages are so complicated you can't easily learn the math skills by using them. But if you learn the math skills in a simpler setting like ACL2, you can apply them to your programming in other languages.

Returning to the question of defining `len1`, a slightly different way to solve this problem comes from thinking like this: "Suppose we want to find the number of elements in `x` by scanning down `x` and counting. Let `i` be the number of elements we've seen so far. Then

when we reach the end of x, the answer is i. Otherwise, we should step past the current element of x, increment i by 1, and keep scanning." This translates into the following code:

```
(defun len2 (x i)
  (if (endp x)
      i
      (len2 (rest x) (+ 1 i))))
```

Note that the function len2 takes two arguments, the current x, and the current i. I claim that (len2 x 0) is the same as (len1 x).

More generally, (len2 x i) is (len1 x)+i, that is, (len2 x i) computes the number of elements of x plus whatever value of i you start with. We'll see how to prove this later.

Functions like len2 are called *accumulator* functions because they have "extra" arguments that accumulate the running answer. Those extra arguments are also called *accumulators*. It is actually a little harder to reason about them than about functions like len1, because they come from thinking about a "process" for solving a problem rather than what the "solution" is. But (len2 x 0) is a mathematically correct answer and is equivalent to all the versions of (len x) we've seen.

My Answer: Finally, what would I turn in if asked to do this problem? I'd do the problem in ACL2 and show the grader that I had done so. I'd also test my answer with the same examples given the problem. (And maybe more if I felt that those examples didn't illustrate all the aspects of my definition.) Here is the transcript of my ACL2 session after defining one of the correct versions of len.

```
ACL2 p>(defun len1 (x)
  (if (endp x)
      0
      (+ 1 (len1 (rest x)))))
Summary
Form:    ( DEFUN LEN1 ...)
Rules:  NIL
Warnings:    None
Time:    0.01 seconds (prove:  0.00, print:  0.00, other:  0.01)
 LEN
ACL2 p>(len1 nil)
0
ACL2 p>(len1 '(A B C))
3
ACL2 p>(len1 '((A 1) (B 2) (C 3) (D 4)))
4
ACL2 p>(len1 '((A B C D E F G H I J K L)))
1
ACL2 p>(len1 45)
0
```

This transcript of my session shows the grader that my definition is syntactically ok and that, whether right or wrong, it at least computes the expected answers on the examples given in the problem.

Note: The above session can't actually be carried out in ACL2 because the function `len` is pre-defined. But you could carry out an analogous session if you changed the name `len` to `my-len`.

• **Question 83**: We call a list whose terminal marker is `nil` a *true list*. Define `true-listp` to recognize true lists. Here are some examples:

```
(true-listp nil) ⟹ t
(true-listp 45) ⟹ nil
(true-listp '(Mon Wed Fri)) ⟹ t
(true-listp '(1 2 3)) ⟹ t
(true-listp '(1 2 3 .  45)) ⟹ nil •
```

Since I don't want you to have to read dot notation, I could rephrase the last test as

```
(true-listp (cons 1 (cons 2 (cons 3 45)))) ⟹ nil •
```

My Thinking: I can think of two obvious ways to define this function. The first way is a direct translation of the definition of "true list:" `x` is a true list if the terminal marker of `x` is `nil`. Suppose I had already defined `terminal-marker` to return the terminal marker of an object. Then here's the "direct translation"

```
(defun true-listp (x)
  (equal (terminal-marker x) nil))
```

This function clearly returns `t` if the terminal marker is `nil` and it returns `nil` if the terminal marker is anything else. So it is a recognizer for the condition "is the terminal marker `nil`?"

But I must define `terminal-marker`. The terminal marker of something is the first non-cons you encounter down the spine. So suppose we're trying to find the terminal marker of `x`. If `x` is empty, `x` *is* the terminal marker. Otherwise, we have an `x` of the form (δ_1 δ_2 ...) where we're looking for the terminal marker at the end of those dots. What do we do? But your friend is there and says "Hey, I can tell you the terminal marker of (δ_2 ...)! It's a." And so, yet again, you say "Cool! I don't know how you did that, but it means my answer is a! Thanks!" So our definition is

```
(defun terminal-marker (x)
  (if (endp x)
      x
      (terminal-marker (rest x))))
```

If we test this with `(terminal-marker '(1 2 3))` we get the correct answer, `nil`. That's reassuring but it would be nice to test it on some object whose terminal marker was not `nil`. I promised I wouldn't expect you to deal the printed forms of such objects. But here

is a test that avoids using the printed form:

```
(terminal-marker (cons 1 (cons 2 (cons 3 45)))) ⟹ 45.
```

So, testing the definition of `true-listp`

```
(true-listp '(1 2 3)) ⟹ t
(true-listp '(1 2 3 .  45)) ⟹ nil
(true-listp (cons 1 (cons 2 (cons 3 45)))) ⟹ nil
```

There's a second way to define `true-listp`: forget about inventing `terminal-marker` and make `true-listp` answer the question itself. If `x` is empty, then it's a true list if it's `nil`. Otherwise, ... "Hey! I can tell you whether `(rest x)` is a true list."

```
(defun true-listp (x)
  (if (endp x)                    ; if we're at the end,
      (equal x nil)               ;  see if x is nil
      (true-listp (rest x))))     ; else, look at rest
```

As before, we could rearrange this in numerous ways and still get the correct answer. Furthermore, the first and second definitions of `true-listp` are equivalent and you'll learn how to prove that (see Question 224, page 176).

Note: The function `true-listp` is pre-defined in ACL2.

• **Question 84**: Define the function `mem` so that `(mem e x)` is `t` if `e` is an element of the list `x` and `nil` otherwise. For example:

```
(mem 2 nil) ⟹ nil
(mem 2 '(1 2 3)) ⟹ t
(mem 2 '(1 3 5)) ⟹ nil
(mem 2 '((A 1) (B 2) (C 3))) ⟹ nil
(mem '(B 2) '((A 1) (B 2) (C 3))) ⟹ t •
```

My Thinking: When you're defining a function with more than one argument you have to decide which argument to reduce. If you're looking for `e` in `x`, the question is do you look for the rest of `e` in `x` or do you look for `e` in the rest of `x`? I don't see how looking for the rest of `e` can help me. Even if I find it somewhere in `x`, it doesn't mean that `e` is in `x`. So I'm going to do my case analysis on `x`: either it's empty or it's not.

When `x` is empty, is `e` an element of it? No. So far we have:

```
(defun mem (e x)
  (if (endp x)
      nil
      __β__))
```

Now what? At β we know that `x` is not empty. So `x` is of the form (δ_1 δ_2 ...), where $n > 0$, `(first x)` is δ_1 and `(rest x)` is (δ_2 ...). How can we solve the problem of whether `e` is

among the δ_i if don't even know how many δ_i there are? Don't think like that!

Your friend can tell you if `e` is among the `(`δ_2 `...)`. So your thinking should be: "If `e` is δ_1, then the answer is `t` because `e` is in `x`; otherwise, I'll ask my friend to solve the smaller problem."

Putting this reasoning into code gives

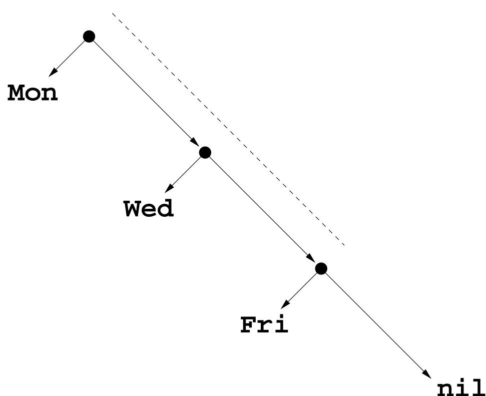
β:
```
(if (equal e (first x))
    t
    (mem e (rest x))).
```

Our answer is thus:

```
(defun mem (e x)
  (if (endp x)
      nil
      (if (equal e (first x))
          t
          (mem e (rest x)))))
```

Thus, `(mem 'Wed '(Mon Wed Fri))` is `t` and `(mem 'Sun '(Mon Wed Fri))` is `nil`.

Is `nil` a member of `(Mon Wed Fri)`? Some students will say yes, because the see a `nil` in this picture:



But that `nil` is not an element! It is the terminal marker. The elements "hang off the spine."

Note that β is the same as "either `e` is `(first x)` or `e` is a member of `(rest x)`." This could be written as β' below:

β':
```
(or (equal e (first x))
```

```
    (mem e (rest x)))
```

I can convince you that β is the same as β' as follows:

β'
=
```
(or (equal e (first x))
    (mem e (rest x)))
```
= {def of}`r (page 49)`}
```
(if (equal e (first x))
    (equal e (first x))
    (mem e (rest x)))
```
= {since we know `(equal e (first x))` is}
on the true branch of an `if` testing it}
```
(if (equal e (first x))
    t
    (mem e (rest x)))
```
=
β

So we could have written `mem` this way:

```
(defun mem (e x)
  (if (endp x)
      nil
      (or (equal e (first x))
          (mem e (rest x)))))
```

and it would be correct. As we've seen, we could also rearrange this using the definitions of `endp`, `first`, `rest`, properties of `if`, etc., and be sure our new code is equivalent to the old code.

The definition of `mem` illustrates an important point: Sometimes to "convert the solution to the smaller problem into the solution of the bigger problem" we have to do some *case analysis* – we have to use `if`. Many students just look for "one thing" they can "do" to the smaller solution to turn it into the bigger one. Don't think that way. Your friend and you are partners and you have to do your part!

Note: The function `mem` is *not* pre-defined in ACL2, but there is a very similar function named `member`. However, I recommend that you define `mem` because it always returns `t` or `nil`; `member` is more complicated and returns "where" it finds `e` in `x`.

• **Question 85**: Recall the discussion of the function `long-day-name` on page 40. This function takes a "short day name" and returns the full name of the day, or `nil` if we can't make sense of the short name. It uses a built-in constant to establish the various abbreviations:

```
((SUNDAY    1 SUN SU)
```

```
(MONDAY    2 MON M)
(TUESDAY   3 TUE TU)
(WEDNESDAY 4 WED W)
(THURSDAY  5 THU TH THUR THURS)
(FRIDAY    6 FRI F)
(SATURDAY  7 SAT SA)).
```

Define `long-day-name`. For example:

```
(long-day-name 4) ⟹ WEDNESDAY
(long-day-name 'WED) ⟹ WEDNESDAY
(long-day-name 'THUR) ⟹ THURSDAY
(long-day-name 'S) ⟹ NIL •
```

My Thinking: One way to solve this problem is so totally hideous I mention it only to discourage you from doing such things:

```
(defun long-day-name (x)
  (if (or (equal x 'SUNDAY)
          (equal x 1)
          (equal x 'SUN)
          (equal x 'SU))
      'SUNDAY
      (if (or (equal x 'MONDAY)
              (equal x 2)
              ...)
          'MONDAY
          ...)))
```

where the function returns `nil` at the very bottom if `x` is not equal to any of the names. This is a terrible way to code because if the database is changed, we might have to inspect every line.

It is much more elegant to imagine that we don't know what's in the database. Elegance matters. The simpler our definition, the more likely it's correct and the easier it is to modify.

So I will choose to write a general purpose function that searches a list of list and finds the first element containing the thing we're looking for, or returns `nil` if it's not there. Let's call the general function `search-for-day-name`. Then my definition of `long-day-name` will be:

```
(defun long-day-name (x)
  (search-for-day-name x
    '((SUNDAY    1 SUN SU)
      (MONDAY    2 MON M)
      (TUESDAY   3 TUE TU)
      (WEDNESDAY 4 WED W)
      (THURSDAY  5 THU TH THUR THURS)
```

```
    (FRIDAY    6 FRI F)
    (SATURDAY  7 SAT SA))))
```

But now my problem is to define a `search-for-day-name`. The basic idea is to look for the first argument, `x`, as an element of an element of second argument, `db`. If the database is empty, then `x` isn't an abbreviation for any day in it and I should return `nil`. Otherwise, I have to determine whether the `first` element of `db` contains `x` and answer appropriately if it does. Otherwise, I can ask my friend to search the `rest` of the database.

```
(defun search-for-day-name (x db)
  (if (endp db)
      nil
      (if α
          β
          (search-for-day-name x (rest db)))))
```

This "glue code" searches down `db` and returns `nil` if I get to the end without ever finding `x`. The test α determines if `x` is an element of the `first` element of `db`. If so, β returns the "long day name" extracted from that element.

How do I determine if `x` is an element of some list? I use the function we already wrote, `mem`! And what is the long day name corresponding to an element of the database? It is the `first` element of that element. So my answer is

```
(defun search-for-day-name (x db)
  (if (endp db)
      nil
      (if (mem x (first db))
          (first (first db))
          (search-for-day-name x (rest db)))))
```

This is a very elegant solution that allows us easily to change the abbreviations or even use the function to resolve other kinds of abbreviations, like month names, nicknames, etc.

There were two keys to making this elegant. The first was deciding to tackle the general problem of searching a list of lists instead of coding up the particular database we were given. The second was recognizing that we already had `mem` for answering the key question of whether `x` is in one of the lists of abbreviations.

The sign of a good programmer is that he or she invents the functions needed to make definitions elegant.

I can't resist, however, showing you the code I'd write. First, I wouldn't invent "`search--for-day-name`" because my database might not be about day names! Since all my function does is determine whether `x` is a member of a member of a list of lists, I'd call it `mem-mem`. I would define it to return the "winning" element, not the first element of the winning element. That way I could let the caller decide what to do with the list of equivalent abbreviations. (Maybe some callers would want the "day number" rather than the long day name.) So my

subfunction would be

```
(defun mem-mem (x db)
  (if (endp db)
      nil
      (if (mem x (first db))
          (first db)
          (mem-mem x (rest db)))))
```

• **Question 86**: Define `nth` so that `(nth i x)` is the $\mathtt{i}^{th}$ element of `x`, where indexing starts at `0`. You may assume that `i` is a natural number that is not too big.

```
(nth 0 '((A 1) (B 2) (C 3))) ⟹ (A 1)
(nth 1 '((A 1) (B 2) (C 3))) ⟹ (B 2)
(nth 2 '(A B C D)) ⟹ C
```

I don't care what value is returned by `(nth 7 '(A B C D))` because `7` is too big an index for that list. •

My Thinking: Once again, we have a two argument function. Will we recurse into `i` or into `x`? We'll consider both choices, starting with `i`.

Since `i` is a natural number, the obvious base case is when `(zp i)` is true. But we know what the $\mathtt{0}^{th}$ element of `x` is: it is `(first x)` if `x` is non-empty. If `x` is empty, we don't care what the function returns and so I'll return `23`! So I've got this so far:

```
(defun nth (i x)
  (if (zp i)
      (if (endp x)
          23
          (first x))
      __β__ )).
```

where β handles the case for `i`> 0. If `x` is empty, then `i` is too big and I'll return `23`. But now what?

Since (`i`-1) is smaller than `i`, we could ask our friend to give us the $(\mathtt{i}-1)^{th}$ element of `x`. But that doesn't help! If we're looking for the 3rd element of `x`, it doesn't help to know what the 2nd element is! But, if `x` is not empty, then the 3rd element of `x` is the 2nd element of `(rest x)`.

$$\overbrace{(\delta_0 \;\; \underbrace{\delta_1 \;\; \delta_2 \;\; \delta_3 \;\; \ldots)}_{\texttt{(rest x)}}}^{\texttt{x}}$$

So if `x` is not empty, we'll ask our friend to give us the $(\mathtt{i}-1)^{th}$ element of `(rest x)`. Remember he or she claims to be able to answer *any* smaller question of this kind. Translating this to code and filling in β gives us:

```
(defun nth (i x)
  (if (zp i)
      (if (endp x)
          23
          (first x))
      (if (endp x)
          23
          (nth (- i 1) (rest x)))))
```

If you test this definition you get these results:

```
(nth 0 '(A B C D E)) ⟹ A
(nth 1 '(A B C D E)) ⟹ B
(nth 3 '(A B C D E)) ⟹ D
(nth 7 '(A B C D E)) ⟹ 23
```

We can clean this definition up by making a different choice for our "don't care" answer. Take the case where `i` is `0`. Why test whether `x` is empty? `(first x)` returns *something* whether `x` is empty or not. We can just skip that first `(endp x)` test and let our "don't care" answer be whatever `first` returns when its argument is empty. Similiarly, if our notion of the "size" of the problem is just the size of `i`, then the problem solved by `(nth (- i 1) (rest x))` is smaller than that solved by `(nth i x)`, whether `x` is empty or not. We're giving our friend a smaller `i` and that is all that matters.

This reasoning gives rise to a new definition. This definition is *not* equivalent to the one above because the "don't care" answer is different. But I like this definition better and so I'll undo the definition above and use this one.

```
(defun nth (i x)
  (if (zp i)
      (first x)
      (nth (- i 1) (rest x))))
```

If you test this definition you get the same results on all the normal tests but a different "don't care" result.

```
(nth 0 '(A B C D E)) ⟹ A
(nth 1 '(A B C D E)) ⟹ B
(nth 3 '(A B C D E)) ⟹ D
(nth 7 '(A B C D E)) ⟹ nil
```

The reason it returns `nil` is that `(first nil)` = `(rest nil)` = `nil`, by the definitions of `first` and `rest`.

Remember that we didn't know whether to recur on `i` or `x` and we were going to consider both choices. We've just finished considering `i`.

If we recur on `x`, we must ask whether it is empty. If so, we want its $\mathtt{i}^{th}$ element, but there

isn't one. Using our new "don't care" choice of `nil`, the answer is `nil`. If `x` is non-empty, then its $\texttt{i}^{th}$ element depends on `i`. If `i` is `0`, then the $\texttt{i}^{th}$ element of `x` is `(first x)`. If `i` is not `0`, we can recur just as before and ask our friend for the $(\texttt{i} - 1)^{th}$ element of `(rest x)`. The translation to code is:

```
(defun nth (i x)
  (if (endp x)
      nil
      (if (zp i)
          (first x)
          (nth (- i 1) (rest x)))))
```

This function has exactly the same behavior as the previous definition. In fact, we can prove they are equivalent.

Note: The function `nth` is pre-defined in ACL2.

• **Question 87**: Define `del1` so that `(del1 e x)` returns a list with the same elements as `x` except that the first occurrence of `e` is missing. If `e` isn't a member of `x`, `del1` should return `x`. You may assume `x` is a true list. The name "`del1`" means "delete one occurrence."

```
(del1 'B '(A B C B D)) ⟹ (A C B D)
(del1 'B '(A C B D)) ⟹ (A C D)
(del1 'B '(A C D)) ⟹ (A C D)
(del1 '(B 2) '((A 1) (B 2) (B 2 3) ((B 2)))) ⟹ ((A 1) (B 2 3) ((B 2)))
(del1 '(B 2) '((A 1) (B 2 3) ((B 2)))) ⟹ ((A 1) (B 2 3) ((B 2))) •
```

My Thinking: Recall `mem`

```
(defun mem (e x)
  (if (endp x)
      nil
      (if (equal e (first x))
          t
          (mem e (rest x)))))
```

The function `del1` will be like `mem`, so I'll use it as a template:

```
(defun del1 (e x)
  (if (endp x)
      α
      (if (equal e (first x))
          β
          γ)))
```

This is a common practice: find a function like the one you want and "modify" it.

There are three cases: α: `x` is empty, β: `x` is a cons whose first element is `e`, and γ: `x` is a

cons whose first element is not `e`. Remember what `(del1 e x)` is supposed to be: `x` with the first `e` deleted.

α: If `x` is empty, `(del1 e x)` is `x` with the first `e` deleted. But there are no `e`'s in the empty list, so we can return `nil` or (equivalently) `x`, since `x` is empty.

β: If `x` is (δ_1 δ_2 ...), where `e` is δ_1, then `x` with the first `e` deleted is just (δ_2 ...), which is `(rest x)`.

γ: If `x` is (δ_1 δ_2 ...), where `e` is not δ_1, then `x` with the first `e` deleted has δ_1 as its first element (because it is not `e`!) and then has the rest of `x` with the first `e` deleted. That "rest" part is just delivered by our friend. We build our answer by using `cons` to add δ_1 back onto the front of that answer.

```
(defun del1 (e x)
  (if (endp x)
      nil                           ; α
      (if (equal e (first x))
          (rest x)                  ; β
          (cons (first x)           ; γ
                (del1 e (rest x))))))
```

If we test it we get the answers expected:

```
(del1 'C '(A B C A B C)) ⟹ (A B A B C)
(del1 'C '(A B A B C)) ⟹ (A B A B C)
(del1 'C '(A B A B)) ⟹ (A B A B)
```

2.15 Practice, Practice, Practice (Assignment 6: 7 days)

• **Question 88**: Define `del*` so that `(del* e x)` returns a list like `x` but with all occurrences of `e` missing. For example

```
(del* 1 NIL) ⟹ NIL
(del* 1 '(1 2 3 1 4)) ⟹ (2 3 4)
(del* 'A '(A A A)) ⟹ NIL
(del* 7 '(1 2 3 1 4)) ⟹ (1 2 3 1 4) •
```

• **QUESTION 89**: The concatenation of two lists `x` and `y` is the list whose first n elements are those of `x` and whose remaining elements are those of `y`, where n is the number of elements in `x`. Define the function `app` so that `(app x y)` is the concatentation of `x` and `y` and demonstrate your definition on the following examples.

```
(app NIL '(1 2 3)) ⟹ (1 2 3)
(app '(1 2 3) NIL) ⟹ (1 2 3)
```

```
(app '(1 2 3) '(4 5 6)) ⟹ (1 2 3 4 5 6)
(app '(1 2 3) '(1 2)) ⟹ • (1 2 3 1 2) •
```

• **Question 90**: Define `no-dups` so that `(no-dups x)` is true if and only iff `x` is a true list containing no duplicate elements. For example

```
(no-dups '(a b c)) ⟹ T
(no-dups '(a b a c)) ⟹ NIL
(no-dups '((a 1) a 1)) ⟹ T
(no-dups '((a 1) (a 1) (b 2)) ⟹ NIL •
```

• **Question 91**: Define `rem-dups` so that `(rem-dups x)` is list with the same elements as `x` but in which no element occurs more than once. I don't care which order the elements are in when you're done. For example

```
(rem-dups '(a b a c b b d)) ⟹ (A B C D)
(rem-dups '(1 1 1 2 2 2)) ⟹ (1 2)
(rem-dups '(1 2 3 4)) ⟹ (1 2 3 4)
(rem-dups '((A 1) (B 2) (A 1) (B 3))) ⟹ ((A 1) (B 2) (B 3))
```

Note that since I don't care which order you list the surviving elements, a correct answer might not pass the tests above. For example, another correct answer to the first evaluation above could be

```
(rem-dups '(a b a c b b d)) ⟹ (A C B D)
```

depending on the algorithm you use. Note also that since we don't distinguish case in symbols, I might have written this last example like this and meant the same thing:

```
(rem-dups '(a b a c b b d)) ⟹ (a c b d) •
```

• **QUESTION 92**: Define the function `rev` so that `(rev x)` is the reverse of the list `x`, and demonstrate your definition on the following examples. For example

```
(rev nil) ⟹ nil
(rev '(A)) ⟹ (A)
(rev '(A B C)) ⟹ (C B A)
(rev '((A 1 2) (B 3 4) (C 5 6)) ⟹ ((C 5 6) (B 3 4) (A 1 2))
```

Hint: If your definition of `rev` behaves like this:

```
(rev '(1 2 3)) ⟹ (((NIL . 3) .  2) .  1)
```

it is *wrong!*. The dots tell you you're consing something onto a non-list. •

• **QUESTION 93**: Define the function `how-many` so that `(how-many e x)` counts the number of times `e` occurs as an element of the list `x`, and demonstrate your definition on the following examples.

```
(how-many 'A '(A B A C A A)) ⟹ 4
(how-many 'Z '(A B A C A A)) ⟹ 0
(how-many 'A '((A 1) (B 2) (A 3))) ⟹ 0
(how-many '(A 1) '((A 1) (B 2) (A 3))) ⟹ 1 •
```

• **Question 94**: One list is said to be a *subset* of another if every element of the first list is an element of the second. Define `subeq` so that `(subeq x y)` checks whether `x` is a subset of `y`, returning `t` or `nil`. By the way, in normal mathematics, this relation is defined on *sets*, not lists, and is written $x \subseteq y$. For example

```
(subeq '(1 3 5) '(1 2 3 4 5 6)) ⟹ t
(subeq '(3 5 1) '(1 2 3 4 5 6)) ⟹ t
(subeq '(3 55 1) '(1 2 3 4 5 6)) ⟹ nil
(subeq '(3 3 3) '(3 2)) ⟹ t •
```

• **Question 95**: The subset relation on lists is transitive. That is, if `x` is a subset of `y` and `y` is a subset of `z`, then `x` is a subset of `z`. Say that in ACL2. •

• **Question 96**: Define `same-elements` so that `(same-elements x y)` is true if lists `x` and `y` have the same elements, i.e., every element in `x` is in `y` and vice versa. Note that two lists can have the same elements and not be equal! In fact, a short list can have the same elements as a long list (because of duplications). For example

```
(same-elements '(1 2) '(2 2 2 1 1 1)) ⟹ t
(same-elements '(1 2 3) '(3 2 1)) ⟹ t
(same-elements '(1 2 2 3) '(3 3 3 2 1)) ⟹ t
(same-elements '(1 2 2 4) '(3 3 3 2 1)) ⟹ nil
(same-elements '(1 2 2 3) '(3 5 3 2 1)) ⟹ nil •
```

• **Question 97**: The *union* of two lists is the list that contains all the elements of each and no other elements. Define `unn` so that `(unn x y)` is the union of `x` and `y`. I don't care what order you list the elements in nor whether you introduce duplications. By the way, in normal mathematics, this operation is defined on sets and is written $x \cup y$. For example

```
(unn '(1 3 5) '(0 2 4)) ⟹ (1 3 5 0 2 4)
(unn '(1 3 5) '(1 2 4)) ⟹ (3 5 1 2 4)
(unn '(A B A) '(B C D)) ⟹ (A B C D)
(unn '(A B) '((A B))) ⟹ (A B (A B))
```

Note that since I don't care about the order or duplications among the elements in the final answer, a correct function could evaluate differently. For example, this would also be acceptable:

```
(unn '(1 3 5) '(0 2 4)) ⟹ (5 3 1 0 2 4) •
```

• **Question 98**: The *intersection* of two lists is the list that contains just the objects that are in both. Define `int` so that `(int x y)` is the intersection of `x` and `y`. I don't care about

order or duplications. By the way, in normal mathematics, this operation is defined on sets and is written $x \cap y$. For example

```
(int '(1 3 5) '(0 2 4)) ⟹ nil
(int '(1 3 5) '(1 2 3)) ⟹ (1 3)
(int '(s m t w t f s) '(s s)) ⟹ (S)
(int '(A B) '((A B))) ⟹ nil
```

Note that since I don't care about order or duplications, your results may vary and still be correct. •

• **Question 99**: The *difference* between two lists is the list that contains the elements of the first that are not in the second. Define `diff` so that `(diff x y)` is the difference of `x` and `y`. I don't care about order or duplications. By the way, in normal mathematics, this operation is defined on sets and is written $x \setminus y$. For example

```
(diff '(1 2 3 4 5 6) '(0 2 4)) ⟹ (1 3 5 6)
(diff '(1 1 2 2 3 3) '(2)) ⟹ (1 1 3 3)
(diff '(A B C) nil) ⟹ (A B C)
(diff '(1 1 2 2) '(2 2)) ⟹ (1 1)
```

Since I don't care about order or duplications, your results may vary and still be correct. •

In the next few questions, I use the notion of a "2-column table." A *2-column table* is a list in which each element is of the form (κ τ), where κ is called a *key* and τ is called a *value*. For example, here is a 2-column table:

```
((A  27)
 (B  14)
 (D  33)
 (A   7)
 (E  45))
```

In the material below I will refer to "table" and mean a "2-column table." You can imagine what I mean by a 3-column table, etc.

Note that that a key may occur twice in a table. We say that the first element of the table with key κ *binds* κ to the value associated with that κ in that element. For example, the element `(B 14)` above binds `B` to `14`. The element `(A 27)` binds `A` to `27`. The second occurrence of the key `A` above is irrelevant. Keys and values can be any kind of object.

It is not a coincidence that we say a "a variable is bound in the environment" and we say that "a key is bound in a table." An environment is just a 2-column table binding variable symbols to their values.

• **Question 100**: Define `tablep` to recognize tables. For example

```
(tablep '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ t
(tablep '((A 27) (B 14 15) (C 33) (A 7) (E 45))) ⟹ nil
```

```
(tablep '((A 27) (B 14) (C) (A 7) (E 45))) ⟹ nil
(tablep '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ t •
```

• **Question 101**: Define `in-tablep` to take `k` and a table `table` and return `t` or `nil` depending on whether `k` is bound in `table`. For example,

```
(in-tablep 'E '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ t
(in-tablep 'D '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ nil
(in-tablep '(A 27) '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ nil
(in-tablep '(B 2) '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ t •
```

• **Question 102**: Define `lookup` to take `k` and a table and to return the value to which `k` is bound in `table`, if `k` is bound. I don't care what value is returned if `k` is not bound. For example,

```
(lookup 'E '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ 45
(lookup 'A '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ 27
(lookup 'B '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ 14
(lookup '(B 2) '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ 14
(lookup 'B '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ 2 •
```

A table is often called a *map* in computer science, because it "maps" keys to values. Given a map, the *domain* is the list ("set") of keys. The `range` is the list ("set") of values. The *image* of `x` in a table `table` is the value associated with `x` if `x` is bound in `table` and is `x` otherwise. (Note: More commonly in mathematics, the "image" of an object not bound in the given table is left undefined.)

• **Question 103**: Define `domain` to take a table and return its domain. (I don't care about the order or duplications.) For example

```
(domain '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ (A B C E)
(domain '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ ((A 1) (B 2) B)
(domain '((A 1) (B 2) (A 3) (B 4) (C 5)))⟹ (A B C)
```

Since I don't care about order or duplications, your results may vary and still be correct. •

• **Question 104**: Define `range` to take a table and return its range. (I don't care about the order or duplications.) For example

```
(range '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ (27 14 33 45)
(range '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ (27 14 2)
(range '((A 1) (B 2) (A 3) (B 4) (C 5)))⟹ (1 2 5)
```

Since I don't care about order or duplications, your results may vary and still be correct. Note however that "duplicate bindings" after the first do not contribute to the range! For example, 7 is not in the range of the first table above, because there is no key that maps to

7. (A is mapped to 27.) •

• **Question 105**: Define `image` to take an object and a table and return the image of the object. For example

```
(image 'C '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ 33
(image 'D '((A 27) (B 14) (C 33) (A 7) (E 45))) ⟹ D
(image '(B 2) '(((A 1) 27) ((B 2) 14) (B 2))) ⟹ 14
(image '(A 3) '(((A 1) 27) ((B 2) 14) (B 2)))⟹ (A 3) •
```

• **Question 106**: Define `make-table` so that given a list of keys and a list of values of the same length, it returns a table in which each key is associated with the corresponding value. For example

```
(make-table '(A B C) '(1 2 3)) ⟹ ((A 1) (B 2) (C 3))
(make-table '(A B A C) '(1 2 7 3)) ⟹ ((A 1) (B 2) (A 7) (C 3)) •
```

• **Question 107**: Define `findpos` so that `(findpos e x)` returns a natural number i such that `(nth` i `x)` is `e`, if `e` is an element of `x`; if `e` is not an element of `x`, return `nil`. Note: Lisp programmers use functions like this to answer "two questions at once:" Is `e` an element of `x`? What's the index of `e` in `x`? Such functions exploit Lisp's ability to use non-Booleans in tests. One might write

```
(if (findpos e x) (+ (findpos e x) ...) ...)
```

in which `(findpos e x)` is used as an expression of both "type Boolean" and "type Nat." For example

```
(findpos 'A '(A B C D E C)) ⟹ 0
(findpos 'C '(A B C D E C)) ⟹ 2
(findpos 'G '(A B C D E C)) ⟹ nil
(findpos '(A 2) '((A 0) (A 1) (A 2) (A 3))) ⟹ 2
```

Note there is some ambiguity in this specification. A correct answer to the second example above could be `5` instead of `2`. •

• **Question 108**: Suppose `x` is a list of lists, like `((a b) (c d e) (x y z))`. Define `add-to-each` so that `(add-to-each e x)` will `cons` `e` onto the front of each element in `x`. For example

```
(add-to-each 'A '(nil nil nil)) ⟹ ((A) (A) (A))
(add-to-each 'A '((1) (2) (3))) ⟹ ((A 1) (A 2) (A 3))
(add-to-each 'B '((A 1) (A 2) (A 3)) ⟹ ((B A 1) (B A 2) (B A 3))
(add-to-each nil '((1) (1 2))) ⟹ ((NIL 1) (NIL 1 2)) •
```

• **QUESTION 109**: Two lists are permutations of each other if you can get one of the lists by rearranging the order of the elements in the other. If `x` is a permutation of `y` then

it follows that `y` is a permutation of `x`. Define `perm` so that `(perm x y)` recognizes whether `x` and `y` are permutations of each other, and demonstrate your definition on the example evaluations given below. **Hint**: One way to do this is to check that every element that occurs in either list occurs the same number of times in both. Another way is to repeatedly find and delete identical elements from the two lists. Here is a suggestive trace of a successful check:

```
(A B B C)  versus  (B A C B)
(  B B C)  versus  (B   C B)
(    B C)  versus  (    C B)
(      C)  versus  (    C  )
(       )  versus  (       )

(A B B C)  versus  (B A C C)
(  B B C)  versus  (B   C C)
(    B C)  versus  (    C C)
```

Success is when both lists are exhausted at the same time. Failure occurs if an element of one list is not an element of the other.

```
(perm '(A B C) '(B C A)) ⟹ t
(perm '(A B C) '(B C A A)) ⟹ nil
(perm '(A B C) '(C A)) ⟹ nil
(perm '(A B A C B) '(A A B B C)) ⟹ t
```

To follow this suggestion you will need to define and use `mem` (page 66) and `del1` (page 72). •

• **Question 110**: How many permutations are there of the list `(a b c)`? •

• **Question 111**: Suppose list `x` contains no duplications. How many permutations of `x` are there? •

In the following questions, you need to know about an ACL2 function we won't use elsewhere. `Lexorder` is like "less than or equal" except instead of just being able to determine if one number is smaller than (or equal to) another, `lexorder` can consider any two ACL2 objects and determine whether the first is smaller than (or equal to) the second. `Lexorder` imposes what is called a "total ordering" on all ACL2 objects.

We discuss orderings later. But an important property of `lexorder` is that if `(lexorder x y)` and `(lexorder y x)`, then `(equal x y)`. That is, if `x` "$\leq$" `y` and `y` "$\leq$" `x`, then `x` is `y`.

`Lexorder` considers numbers smaller than characters, characters smaller than strings, strings smaller than symbols, and symbols smaller than lists; within those categories, `lexorder` uses an alphabetic-like ordering. So `1` comes before `ALPHA`, which comes before `Monday`, which comes before `(APPLE)`. That is `(lexorder 1 'alpha)` is `t`, as are `(lexorder 'alpha 'monday)` and `(lexorder 'monday '(APPLE))`. `(Lexorder 'BRAVO 'ALPHA)` is `nil`.

• **QUESTION 112**: Define `ordp` so that `(ordp x)` determines whether the list `x` is in

weakly ascending order, where the order is determined by `lexorder`. See the paragraphs above this question for a discussion of that function. Demonstrate your definition on the following examples.

```
(ordp '(1 2 3 4)) ⟹ t
(ordp '(1 1 1 2 2 4 5)) ⟹ t
(ordp '(1 2 2 4 3 5)) ⟹ nil
(ordp '(ALPHA BRAVE BRAVO CHARLIE DOG)) ⟹ t
(ordp '(ALPHA BRAVO BRAVE CHARLIE DOG)) ⟹ nil •
```

• **Question 113**: How many ordered (in the sense of `ordp`) permutations of `x` are there? See Question 109 (page 78) for the definition of `perm`. •

• **QUESTION 114**: Define `insert` so that `(insert e x)` returns the list `x` with `e` added as a new element. In addition, if `x` is ordered (by `ordp` from Question 112, page 79) then so is `(insert e x)`. Demonstrate your definition on the following examples.

```
(insert 3 '(1 1 2 2 4)) ⟹ (1 1 2 2 3 4)
(insert 'BRAVE '(ALPHA BRAVO DOG)) ⟹ (ALPHA BRAVE BRAVO DOG)
(insert 'ZEBRA '(ALPHA BRAVO DOG)) ⟹ (ALPHA BRAVO DOG ZEBRA) •
```

• **QUESTION 115**: Define `isort` so that `(isort x)` takes an arbitrary list and returns a rearrangement of it that is ordered according to `ordp` (Question 112, page 79). Demonstrate your definition on the following examples.

```
(isort '(2 1 2 4 1)) ⟹ (1 1 2 2 4)
(isort '(jane tom susan mary fred)) ⟹ (FRED JANE MARY SUSAN TOM)
(isort '(BRAVE DOG ALPHA BRAVO))  ⟹ (ALPHA BRAVE BRAVO DOG)
(isort '(GREEN 23 (A 1) 12 (A 44)) ⟹  (12 23 GREEN (A 1) (A 44)) •
```

2.16 Abbreviations (Assignment 6 cont'd)

In this section I introduce some abbreviation conventions. When I say something is a "term" I mean it is a term after these conventions have been applied.

We allow `and`, `or`, `+`, and `*` to take more than 2 arguments. When they are given more than 2 arguments, "right-associate" the function. For example

abbreviated term	*actual term*
`(and a b c d)`	`(and a (and b (and c d)))`
`(or a b c)`	`(or a (or b c))`
`(+ i j k m)`	`(+ i (+ j (+ k m)))`
`(* 2 i j)`	`(* 2 (* i j))`

`(list` x_1 x_2 ... x_k`)` is an abbreviation for

(cons x_1
      (cons x_2
            ...
            (cons x_k nil)...))

Thus `(list a b c)` is an abbreviation for `(cons a (cons b (cons c nil)))`. `(list)` is an abbreviation for `nil`.

(let ((v_1 a_1) ... (v_k a_k)) *body*), where the v_i are distinct variable symbols and the a_i and body are terms, is an abbreviation for the term obtained from *body* by simultaneously replacing all the v_i by the corresponding a_i.

So

```
(let ((x (+ a b))
      (y (* 2 x)))
  (list x y))
```

is an abbreviation for

```
(list (+ a b) (* 2 x))
```

which is an abbreviation for

```
(cons (+ a b) (cons (* 2 x) nil))
```

Note that in expanding a `let`-form we replace the variables *simultaneously*, not sequentially. If we did it sequentially,

```
(let ((x (+ a b))
      (y (* 2 x)))
  (list x y))
```

would incorrectly expand to

```
(list (+ a b) (* 2 (+ a b))).
```

Note the difference.

• **Question 116**: What is the value of `(list 'mon 'tue 'wed 'thu 'fri)`? •

• **Question 117**: What is the first of `(list x y z)`? •

• **Question 118**: What is the rest of `(list x y z)`? •

• **Question 119**: Can `(list x y z)` be the same as `(cons u v)`? If so, what must be the values of `u` and `v`? If not, why not? •

• **Question 120**: Can `(list x y)` and `(list u v w)` be the same? If so, what must be

the values of `x` and `y`? If not, why not? •

• **Question 121**: What is value of `(list (list 'name "Susan") (list 'income 65000) (list 'city 'austin))`? •

• **Question 122**: What term does `(and p q r)` abbreviate? •

• **Question 123**: What term does `(and (or p q r) u v)` abbreviate? •

• **Question 124**: What term does this abbreviate?

```
(let ((x (+ a b))
      (y (f 23)))
   (+ (* x x) y)) •
```

Most of the conventions above are implemented with "macros." The ACL2 user can introduce new macros but we don't discuss them here.

3

The Need for Logic (Assignment 7: 2 days)

This course is not about programming. It is about symbolic logic. Symbolic logic is mankind's attempt to codify human thought: a system of rules with which we can deduce truths.

We're interested in determining whether a formula is "always true." By that we mean that it evaluates to true no matter what values the variables have.

A formula that is always true is said, by logicians, to be *valid*.[1]

It is possible to show that a formula is valid by proving it: by showing how to derive it from valid "axioms" using validity-preserving "rules of inference." Axioms are formulas we just assume to be always true; rules of inference are formula-transformers with the property that they produce valid formulas from valid formulas. So anything produced by rules of inference from axioms is valid.

3.1 An Informal Proof (Assignment 7 cont'd)

Remember the function `app` from Question 89, page 73? It concatentates two lists. To remind you, `(app '(1 2 3) '(4 5 6))` evaluates to `'(1 2 3 4 5 6)`. Here is a formula about `app` that is always true:

γ:

```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a)))
```

Call the formula γ. In γ, `a` and `b` are just variables. When I say γ is always true I mean it will evaluate to true no matter what Lisp values we plug in for `a` and `b`. According to γ, the first element of `(app a b)` is either the first element of `b` or the first element of `a`, depending on whether `a` is empty. I hope your intuition tells you that γ is valid (i.e., true

[1] Non-logicians are sloppy on this issue. They will say, for example, "the formula '$x = x$' is true" meaning that it is true for all values of x. The logician reserves the adjective "true" to refer to a particular evaluation of a formula. I'll tend to say "always true" or "valid" unless I'm referring to a formula with no variables. So I might say "the formula '2+2=4' is true."

for all values of `a` and `b`).

We can test the formula by binding `a` and `b` to particular values and evaluating the formula:

```
ACL2 p>(let ((a '(1 2 3))
             (b '(4 5 6)))
         (equal (first (app a b))
                (if (endp a)
                    (first b)
                    (first a))))
T
ACL2 p>(let ((a nil)
             (b '(Mon Tue Wed Thu Fri)))
         (equal (first (app a b))
                (if (endp a)
                    (first b)
                    (first a))))
T
ACL2 p>(let ((a 77)
             (b 15))
         (equal (first (app a b))
                (if (endp a)
                    (first b)
                    (first a))))
T
```

But we cannot test *all* of the possibilities because there are infinite number of them! So how can we possibly show that γ is always true? The answer is we can *prove* it.

To prove it, I need to tell you my definition of `app`:

```
(defun app (x y)
  (if (endp x)
      y
      (cons (first x)
            (app (rest x) y)))).
```

I hope my definition is equivalent to your solution to Question 89, page 73. Here's my thinking: `app` has to be recursive because I don't know how long the list I'm constructing will be. I don't know right away whether to recur on `x` or `y` but just trying `x` first showed me a solution. If `x` is empty, then concatenating it to `y` will produce `y`. Otherwise, `x` is (δ_1 $\ldots_x$) and `y` is ($\ldots_y$). I'm using "$\ldots_x$" to mean the elements in `(rest x)` and "$\ldots_y$" to mean the elements in `y`. I want to produce (δ_1 $\ldots_x$ $\ldots_y$). So I use recursion to concatenate `(rest x)` to `y`, producing ($\ldots_x$ $\ldots_y$) and then I use `cons` to add the first element of `x`, δ_1, onto the front of that to get my final answer.

A definition, such as of `app` above, is an equality that is valid, i.e., it is a "*Truth*," true for all values of the variables.

Truth 1:
```
(app x y)
 =
(if (endp x)
    y
    (cons (first x)
          (app (rest x) y)))).
```

My goal is to convince you that γ is also a *Truth*, where

γ:
```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a)))
```

My plan is to *transform* it replacing equals by equals until I get `t`. This is sometimes called *rewriting* and it is perhaps the most frequently used rule of inference. It is often denoted in proofs with the justification "{rewriting with α}," where α is the name of some truth used to transform the formula. Even more often is denoted simply as "{α}."

The basic idea in rewriting is to use a previously established Truth, say the equality called *Truth 1* above, to replace a subterm of our goal by another subterm with the intention of eventually reducing the goal to something obviously true. One rewrite step requires at least four mini-steps: (a) choose a subterm of the goal as your target, (b) choose a known equality, (c) instantiate the variables of the known equality so that its left-hand side matches the target, (d) replace the target in the goal by the instantiated right-hand side of the known equality. (I've just described substituting right for left, but you can substitute left for right too.)

My first step in proving γ will be to (a) choose the `(app a b)` as my target, (b) choose *Truth 1* as the equality I will use, match the target to the left-hand side of the equality, `(app x y)`, by letting `x` be `a` and `y` be `b`, and (d) replace the target in γ by that instance of the right-hand side of the equality. I know *Truth 1* is always true, so it is true for the particular choices I made for `x` and `y`. Thus, the target is equal to the instantiated right-hand side of the equality. So if I transform γ to γ_1 this way, I know they're equal. Notice that if the equality I use is the definition of `app`, this step is the symbolic counterpart of function call are evaluated: replace the call by the body, with the formals replaced by the actuals.

γ_1:
```
(equal (first (if (endp a)
                  b
                  (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
```

I've underlined the instantiated body of `app`. Everything else in γ_1 is identical to the corresponding part of γ. Since `(app a b)` is equal to the instantiated body, we haven't changed the value of γ. So $\gamma = \gamma_1$.

Because the underlining is distracting, I'm not going to use underlining in the subsequent transformations.

Now here's another truth.

Truth 2:
`(first (if x y z))` = `(if x (first y) (first z))`.

Intuitively, when an `if` is delivering its value to `first`, the same result could be produced by moving the `first` into the `if` and applying it to which ever answer the `if` will return.

We can rewrite with *Truth 2*, exactly analogous to the way we did with the definition of `app`. (a) Choose the target:

```
(first (if (endp a)
           b
           (cons (first a)
                 (app (rest a) b)))).
```

(b) Choose the equality *Truth 2*, with the left-hand side `(first (if x y z))`. (c) Match the two, instantiating the variables of the equality to make its left-hand side be identical to the target. In this case, we let `x` be `(endp a)`, we let `y` be `b`, and we let `z` be the `cons`-expression above. Thus, we know that the target is equal to:

```
(if (endp a)
    (first b)
    (first (cons (first a)
                 (app (rest a) b)))).
```

So mini-step (d) is to replace the target with the term above to get γ_2.

γ_2:
```
(equal (if (endp a)
           (first b)
           (first (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
```

We know that $\gamma_1 = \gamma_2$.

Finally, we know

Truth 3:

`(first (cons x y))` = `x`.

This just says applying `first` to an object created by `cons` returns the first argument supplied to the `cons`. Using *Truth 3*, we can rewrite γ_2, taking the four mini-steps illustrated above, to get

γ_3:
```
(equal (if (endp a)
           (first b)
           (first a))
       (if (endp a)
           (first b)
           (first a)))
```

We know that $\gamma_2 = \gamma_3$. But look! Now the `equal` in γ_3 is being applied to two identical expressions. This brings to mind another truth:

Truth 4:
`(equal x x)` = `t`.

This says that the function `equal` returns `t` when its arguments are identical.

Using *Truth 4*, we can rewrite γ_3 to `t`.

Since $\gamma = \gamma_1 = \gamma_2 = \gamma_3 =$ `t`, we know $\gamma =$ `t`.

This is a *symbolic proof*. We are not just *testing* our claim. We proved it. We wrote our claim as a formula and then we manipulated the formula without knowing anything about `a` and `b`. So γ holds for all possible values of `a` and `b`. Since γ is always true, we can call it a *Truth* and use it in the future to establish other *Truths*.

The really cool thing about mathematical logic is that it is a framework in which you can deduce truths about an infinite number of objects while doing a finite amount of work. Furthermore, the total number of ideas you have to master is quite small. In the proof above, all we needed was (a) some initial *Truths*, the rule that says any instance of a *Truth* is a *Truth*, and substitution of equals for equals. A few more rules are enough to let us deduce anything that can be deduced. That is what I meant when I said that mathematical logic is mankind's attempt to codify human thought.

Here is a succinct description of the proof just given:

Truth 5:
```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a)))
```

Proof: We will transform the above formula to `t`.

```
(equal (first (app a b))
```

```
       (if (endp a)
           (first b)
           (first a)))
=                                                          {Truth 1}
(equal (first (if (endp a)
                  b
                  (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
=                                                          {Truth 2}
(equal (if (endp a)
           (first b)
           (first (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
=                                                          {Truth 3}
(equal (if (endp a)
           (first b)
           (first a))
       (if (endp a)
           (first b)
           (first a)))
=                                                          {Truth 4}
t
□
```

It should be clear that if I can prove that `(equal` α β`)` $=$ `t`, then I could also prove $\alpha = \beta$. We have to use `equal` if we're writing an ACL2 term or definition, but when I'm just giving a mathematical argument I tend to use = instead. So I could have written the truth and its proof this way. Note that I always start my proof with a brief sketch of what I'm going to do.

Truth 5:

```
(first (app a b))
=
(if (endp a) (first b) (first a)))
```

Proof: We will transform the left-hand side, *lhs*, to the right-hand side, *rhs*.

lhs
=

```
(first (app a b))
=                                                      {Truth 1}
(first (if (endp a)
           b
           (cons (first a)
                 (app (rest a) b))))
=                                                      {Truth 2}
(if (endp a)
    (first b)
    (first (cons (first a)
                 (app (rest a) b))))
=                                                      {Truth 3}
(if (endp a)
    (first b)
    (first a))
=
rhs
□
```

This second proof of *Truth 5* is easier to write out because I don't have to keep writing the right-hand side.

Finally, it is more convenient to give names to our *Truths* rather than just number them. I would give the name "`first-app`" to *Truth 5* because the name reminds me that I can rearrange an expression of the form `(first (app ...))`. I would typically refer to *Truth 1* as "def `app`" meaning "definition of `app`." I would refer to *Truths 2* and *3* as `first-if` and `first-cons`.

• **QUESTION 125**: Fill in the nine blanks to complete the proof below. You will not need ACL2 to do this homework. But be extra careful to write well formed terms! In your answer, just list the numbers from 1 to 9 and write what belongs in the corresponding blank. *Some relevant truths that you may use are listed below the proof.*

Theorem:
`(true-listp (rev a))`

Proof:

```
(true-listp (rev a))
=                                                      {def rev}
(true-listp ____1____)
=                                                      {____2____}
(if (endp a)
    (true-listp nil)
    (true-listp (app (rev (rest a)) (cons (first a) nil))))
=                                                      {Computation}
```

```
(if (endp a)
    t
    (true-listp (app (rev (rest a)) (cons (first a) nil))))
=                                                {true-listp-app}
(if (endp a)
    t
    ____3____)
=                                                {____4____}
(if (endp a)
    t
    (if (endp (cons (first a) nil))
        (equal (cons (first a) nil) nil)
        (true-listp (rest (cons (first a) nil)))))
=                                                {def endp }
(if (endp a)
    t
    (if (not (consp (cons (first a) nil)))
        (equal (cons (first a) nil) nil)
        (true-listp (rest (cons (first a) nil)))))
=                                                {____5____}
(if (endp a)
    t
    (if (not t)
        (equal (cons (first a) nil) nil)
        (true-listp (rest (cons (first a) nil)))))
=                                                {Computation}
(if (endp a)
    t
    (if nil
        (equal (cons (first a) nil) nil)
        (true-listp (rest (cons (first a) nil)))))
=                                                {____6____}
(if (endp a)
    t
    (true-listp (rest (cons (first a) nil))))
=                                                {rest-cons}
(if (endp a)
    t
    (true-listp ____7____))
=                                                {Computation}
(if (endp a)
    t
    ____8____)
=                                                {____9____}
t
```

□

For the purposes of this question, you may assume the following truths. The first three are definitions of `app`, `rev`, and `true-listp`.

Def `app`:
```
(app x y)
=
(if (endp x)
    y
    (cons (first x)
          (app (rest x) y)))
```

Def `rev`:
```
(rev x)
=
(if (endp x)
    nil
    (app (rev (rest x))
         (cons (first x) nil)))
```

Def `true-listp`:
```
(true-listp x)
=
(if (endp x)
    (equal x nil)
    (true-listp (rest x)))
```

```
if-nil
(if nil x y) = y
```

```
if-x-y-y:
(if x y y) = y
```

```
consp-cons:
(consp (cons x y)) = t
```

```
true-listp-if:
(true-listp (if x y z)) = (if x (true-listp y) (true-listp z))
```

```
true-listp-app:
(true-listp (app x y)) = (true-listp y)
```

You should read these truths carefully; I believe you'll agree that, intuitively, they are always true. •

3.2 Implication (Assignment 7 cont'd)

Another property of `app` is that if `a` is empty, then `(app a b)` is `b`. I hope your intuition agrees that this is always true.

• **Question 126**: Write an ACL2 term to capture the claim that "if `a` is empty, `(app a b)` is `b`." That is, we want an ACL2 term that is `t` if the claim is true and `nil` if the claim is false. This is called *formalizing* the claim. •

This is a rhetorical question because I want to answer it and discuss the answer! In general, we're trying to formalize "if α is true, then β is true." But the English might have been rephrased "β is true when α is true", or "β is true provided α is true" and any number of other ways.

Clearly, the hypothesis α, "`a` is empty," is formalized as `(endp a)`. The conclusion, β, "`(app a b)` is `b`" is captured by `(equal (app a b) b)`, since I'm trying to write an ACL2 formula. But how do we glue them together?

I hope you remember the discussion of `implies` from page 51, because that is what we need. "if `a` is empty, then `(app a b)` is `b`" is formalized by "`(implies (endp a) (equal (app a b) b))`". Call this formula γ.

I claim that γ is a *Truth*: true no matter what `a` and `b` are. Remember the discussion on page 51. Suppose γ is a *Truth*. What does it tell us for a non-empty `a`? It tells us nothing, because such an `a` makes the hypothesis false, so the implication is vacuously true.[2] That's as it should be because in this statement we don't care about non-empty `a`.

A useful fact about implication is that it is transitive. If α implies β and β implies γ, then α implies γ. As a formula:

Truth `trans-implies:`

```
(implies (and (implies a b)
              (implies b c))
         (implies a c))
```

[2]To students still troubled by the fact that `(implies nil nil)` is `t`, realize that if we changed the definition of `implies` so that it returned `nil` here, γ wouldn't be a *Truth* anymore!

This can be proved by using a more basic fact mentioned in the next question.

• **Question 127**: : If it's true that `a` implies `b`, and `a` is true, then `b` is true. Formalize that as an ACL2 formula. •

The transitivity of `implies` can be proved by reasoning as follows: `a` is either `nil` or not. If it's `nil`, then the transitivity formula is easy to prove. On the other hand, if `a` is true, then since `(implies a b)` is true, we know `b` is true. But since `(implies b c)` is true, we know `c` is true. But that means `(implies a c)` is true – because both `a` and `c` are true. This is just a proof sketch that will guide the construction of a more careful proof later.

This is sometimes called *chaining* implications together.

3.3 Proving Another Simple Property (Assignment 7 cont'd)

Let's prove that if `a` is empty, then `(app a b)` is `b`. This kind of conjecture is called an *identity* because it tells us a condition under which `(app a b)` is identical to one of its arguments.

Truth `app-identity:`
γ:
```
(implies (endp a)
         (equal (app a b) b))
```

Again, we can expand the call of `(app a b)`:

γ_1:
```
(implies (endp a)
         (equal (if (endp a)
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
```

Notice we have a hypothesis of `(endp a)` and another `(endp a)` buried in the conclusion. That second `(endp a)` can be replaced by `t`. Look at it this way: Either `(endp a)` is `t` or `nil`. If `(endp a)` is `nil` then γ is `t` because `implies` is `t` when its hypothesis is false. Since we're trying to prove γ is `t`, we're done with that case. In short, in a proof of an implication, we don't have to consider the cases where the hypotheses are false.

That leaves the case where `(endp a)` is `t`. But if it's `t`, we replace the second `(endp a)` above with `t`, just as though we had the equation `(endp a)`= `t`. This is usually called *using a hypothesis* or denoted in a proof with "*by hyp.* It is a another basic rule of inference and we will describe it in detail in the next chapter. Using it here we get γ_2.

γ_2:

```
(implies (endp a)
         (equal (if t
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
```

Here is another obvious

Truth `if-t`:
`(if t x y)` = `x`.

Using `if-t`, we can transform γ_2 to

γ_3:
```
(implies (endp a)
         (equal b
                b))
```

which we can then transform to

γ_4:
```
(implies (endp a)
         t)
```

by what we called *Truth 4*, more commonly known as the *Reflexivity of Equality.*

Finally, here is a truth about `implies`:

Truth `implies-t`:
`(implies x t)` = `t`.

We'll prove this later. But the intuition is the same as before: `x` is either `nil` or it's not. If `x` is nil, then `(implies x t)` reduces to `t` by computation. If `x` is not `nil`, then since `x` is just tested by `implies`, we can replace it by any non-`nil` value and we choose to replace it by `t`. But `(implies t t)` evaluates to `t`. Since `(implies x t)` reduces to `t` in both cases and since the two cases we chose ("`x` is nil or it's not") cover all the possibilities, `(implies x t)` is always equal to `t`. This style of reasoning is enshrined in another rule of inference, called "proof by cases" or just "cases."

Thus, we've shown $\gamma = \gamma_1 = \gamma_2 = \gamma_3 = \gamma_4 =$ `t`. Again, no testing was used to establish an interesting fact about `app` for an infinite number of inputs.

We could write this proof down as follows:

Truth `app-identity`:
```
(implies (endp a)
         (equal (app a b) b))
```

Proof: I will transform the formula to `t`.

```
(implies (endp a)
         (equal (app a b) b))
=                                                            {def app}
(implies (endp a)
         (equal (if (endp a)
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
=                                                            {hyp}
(implies (endp a)
         (equal (if t
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
=                                                            {if-t}
(implies (endp a)
         (equal b b))
=                                                            {Refl of Equality}
(implies (endp a)
         t)
=                                                            {implies-t}
t
```

□

Another way to write the proof avoids having to write the `(implies (endp a) (equal ... b))` over and over again. The idea behind this style of presentation is that I identify my hypotheses and then just work on the conclusion.

Truth `app-identity`:

```
(implies (endp a)
         (equal (app a b) b))
```

Proof: I transform the *lhs* of the conclusion to the *rhs*, given the *hyp*.

hyp: `(endp a)`

```
lhs
=
(app a b)
=                                                            {def app}
(if (endp a)
    b
    (cons (first a)
```

```
          (app (rest a) b)))
=                                                    {hyp}
(if t
    b
    (cons (first a)
          (app (rest a) b)))
=                                                    {if-t}
b
=                                                    {implies-t}
rhs
□
```

I have now described the proof of `app-identity` three times. But I've only described *one* proof. My logical reasoning has been the same in each description; but the descriptions have changed. You're free to use whatever descriptive style you wish. Just remember that a proof is intended to bring the reader along; the reader should be able to easily grasp how you got from one formula to the next and be sure that every step is correct. Finding good descriptions of your proofs is important – especially if you're being graded!

3.4 Replacement of "Equals" by "Equals" (Assignment 8: 5 days)

We've been using replacement of equals by equals. But recall that (`iff` α β) tests propositional *equivalence*: α and β are both `nil` or both non-`nil`. In some situations it is possible to do replacements of propositional equivalents by propositional equivalents. For example, because of the peculiar non-Boolean definition of `and` in Lisp, `(and p q)` is not necessarily equal to `(and q p)`. But you can clearly rewrite `(if (and p q) ... ...)` to `(if (and q p) ... ...)` without changing the value of the `if`-expression. Why? Because `(iff (and p q) (and q p))` is true and the `and`-expression above is only tested by the `if`-expression: only the propositional values matter.

• **Question 128**: In the following expression you can replace one of the `(and p q)` expressions by `(and q p)`, but not the other one. Given an example evaluation that shows that replacing the wrong occurrence of `(and p q)` changes the value of the `if`-expression.

```
(if (and p q)
    (cons (and p q) nil)
    23)  •
```

We are very interested in propositional equivalences, like the commutativity of `and`, precisely because these equivalences allow us to simplify formulas. But as the previous question question makes clear, you can replace propositional equivalents only in certain situations.

Roughly speaking, you can rearrange a propositional expression using propositional equivalences, provided you only care about the truthvalue – not the actual value – of the expression.

I make this precise below. Please pay attention to the difference between "equal" and "propositionally equivalent." If I say "α and β are *equal*," or "α and β are *identical*," or "α is β," etc., I mean `(equal` α β`)` is true. But if I say α and β are *propositionally equivalent*, I mean `(iff` α β`)` is true.

We say a particular occurrence of a subterm α is *used propositionally* if the occurrence is (i) a formula to be proved, (ii) the test of an `if`-expression, (iii) the true- or false-branch of an `if`-expression that is used propositionally, or (iv) an argument to a call of a propositional function and that call is used propositionally.

Below, assume we're trying to prove the formula shown. In all the examples, α is used propositionally.

```
1.  (implies (and (not α) ...) ...)
2.  (iff α ...)
3.  (implies ...
             (equal (if α
                         β
                         ...)
                    ...))
4.  (implies ...
             (iff (if α
                       β
                       ...)
                  ...))
```

Let's look closely at example 1 above. The occurrence of α shown is an argument to the propositional function `not`. That means that occurrence of α is being used propositionally *if* the containing `not`-expression is being used propositionally. But the `not`-expression is an argument to the propositional function `and`, which means the `not`-expression is being used propositionally if the `and`-expression is being used propositionally. But the `and`-expression is an argument to the propositional function `implies`, which means the `and`-expression is being used propositionally if the `implies`-expression is being used propositionally. But the `implies`-expression is something to be proved, so it is being used propositionally.

In example 3, the α occurs as the test of an `if`, so it is being used propositionally. But look at the occurrence of β. It occurs in the true-branch of that `if`. That *would be* a propositional usage if the `if`-expression were being used propositionally, but the `if`-expression is not being used propositionally! That `if`-expression is returning its value to `equal`, which is not propositional. So β is not being used propositionally in example 3.

However, in example 4, both α and β are used propositionally. In particular, the β is the true-branch of an `if`-expression that is being used propositionally in the `iff` expression.

Whether we can replace α in some larger formula γ by some propositional equivalent depends on *where* the α occurs in γ, not on α itself.

For example, recall `findpos` (page 78) which returns the index, if any, of an item in a list, or `nil` if the item is not an element. We might wish to prove this:

γ:
```
(implies (findpos e x)
         (< (findpos e x) (len x)))
```

Note that `(findpos e x)` occurs twice here. But only the first occurrence is being used propositionally. Using this theorem:

```
(iff (findpos e x)
     (mem e x))
```

I could rewrite the formula γ to

γ':
```
(implies (mem e x)
         (< (findpos e x) (len x)))
```

but not to

γ'':
```
(implies (findpos e x)
         (<  (mem e x) (len x))).
```

So whether we can replace propositional equivalents depends on where the occurrence is in the larger formula, not on the two equivalents.

To be completely formal I would have to define what an *occurrence* is. Technically, an occurrence of α in γ is an *address* in γ at which you find the subterm α. I am not going to define exactly what an address is, but it is easy to illustrate how I could. For example, I could say that an address is a subterm and a natural number, $(\alpha\ n)$, pointing to the nth occurrence of α in the larger expression. Or I could say that an address is a sequence of numbers indicating how to "navigate" down to the subterm in question; for example, the second occurrence of the `findpos`-expression in γ would be `(2 1)`, indicating that it is in the 1st argument of the 2nd argument of γ.

Regardless of how I represent addresses, there is a critical restriction I impose on them: while an address may point to a constant in γ, there shall be no legal address that points *into* the substructures of a constant. For example, there is no address that points to the symbol `Wed` in `(mem e '(Mon Wed Fri))`. I shall allow an address to point to the `'(Mon Wed Fri)` but not into it.

Given some address π in γ, then I will speak of *replacing* the occurrence of the subterm at π in γ by some other term. Again, I could precisely define this but it should be clear. For example, if π points to the second `(f x)` in `(g (f x) (h (f x) y))` and I say "replace the subterm at π by `b`," then the term I obtain is `(g (f x) (h b x))`.

Addresses never point into constants! For example,

```
(mem ele '(I J K L M N))
     ↑π1 ↑π2       ↑π3
```

address π_1 points to the variable `ele` and address π_2 points to the constant expression `'(I J K L M N)`. But there is no legal address π_3 that points to the `M` within that constant! This prevents us from using "replacement of equals by equals" to accidentally change a constant or mistake a constant for a variable. For example, suppose I am proving a formula that has the hypothesis that `M` is equal to `23`. If I had an address for the `M` in the constant above, I might use it to replace that `M` with `23`:

```
(mem ele '(I J K L 23 N))
```

which would be a logical mistake.

We say that an address *admits propositional replacement* if the subterm there is being used propositionally, as defined above.

Having said this much about addresses, I'm not going to bother explaining how to represent or use them because I trust that you understand enough for our purposes. If I want to indicate a particular subterm in some specific formula I'll just describe it ("the second `(f x)`") or point to it at the board, or mark it some way in the display.

• **Question 129**: Is α being used propositionally in γ below? You may assume γ is being used propositionally. If α is not being used propositionally, demonstrate with two ACL2 evaluations that it is possible to replace α by something that is propositionally equivalent and get a different result. You will have to choose values for the other Greek letters.

```
γ:
(and ψ (or φ (iff α δ))) •
```

• **QUESTION 130**: I have labeled each address in γ below. An arrow pointing to an open parenthesis denotes the address of the subterm that starts with that parenthesis. An arrow pointing to a variable means the occurrence of that variable. List the numbers of the addresses admitting propositional replacement. (I.e., if you want to say the addresses in question are π_2 and π_8, just say 2 and 8.) Assume you know nothing about the function symbols `f`, `g`, and `h`. Just use your word processor to prepare list of i such that π_i admits propositional replacement.

```
γ:
(implies (and (consp x)
↑π1      ↑π2  ↑π3    ↑π4
              (or (stringp s)
              ↑π5 ↑π6        ↑π7
                  (natp s)))
                  ↑π8   ↑π9
         (natp (if (and (f x) (g x))
         ↑π10  ↑π11 ↑π12 ↑π13 ↑π14↑π15↑π16
                    (h (f x))
                    ↑π17↑π18↑π19
                    x)))
                    ↑π20
```

3.5 What's Going On Here? (Assignment 8 cont'd)

Logic is mankind's attempt to formalize how we reason. There are several key ideas.

First, we will only prove things that we can write as *formulas* in some fixed notation. For the moment in this course, our notation is ACL2.

Second, we are told some basic truths, expressed as formulas. These basic truths are called *axioms*. The axioms include all the definitions. We're free to add new definitions provided we follow certain rules. Every time we add a new definition, we get a new axiom.

Third, we are given some basic *rules of inference*. These are carefully described transformations on formulas. They have a key property: if used to transform a formula that is always true, they produce a formula that is always true. The most basic rule of inference is that if $\alpha = \beta$ then you can replace α by β or vice versa. Notice that if we start with axioms (which are always true) and transform them with rules of inference, we get formulas that are always true. These formulas, called *theorems*, can then participate in our transformations.

This is what we've been doing when we wrote things like:

Truth `app-identity`:

```
(implies (endp a)
         (equal (app a b) b))
```

Proof: I will transform the formula to `t`.

```
(implies (endp a)
         (equal (app a b) b))
=                                                    {def app}
(implies (endp a)
         (equal (if (endp a)
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
=                                                    {hyp}
(implies (endp a)
         (equal (if t
                    b
                    (cons (first a)
                          (app (rest a) b)))
                b))
=                                                    {if-t}
(implies (endp a)
         (equal b b))
=                                                    {Refl of Equality}
(implies (endp a)
         t)
```

= {implies-t}
`t`
□

except that these proofs should be read backwards, from bottom to top: You may think of `t` as an axiom. Then we transform it to an equivalent formula, `(implies (endp a) t)`. So that formula must be a truth. Then we transform it to ..., and so on. Eventually we reach the goal formula. If the axioms we used are all truths and the rules of inference preserve truth, then the goal formula is a truth too.

3.6 Recognizing and Using Truths (Assignment 8 cont'd)

> Testing shows the presence, not the absence of bugs – *Edsger W. Dijkstra, April 1970.*

You have spent over a dozen years learning the truths of arithmetic. You know $x+y = y+x$, $x \times 1 = x$, $x \times (y/x) = y$ when $x \neq 0$, and $(a+b) \times x = (a \times x) + (b \times x)$. etc.[3]

Of course, our intuitions are often wrong about truths. For example, many people would say that $x \times (y/x) = y$ and leave off the crucial hypothesis that x is non-0. Many people would say $0^n = 0$ when n is a natural, when in fact, $0^0 = 1$. You can confirm that non-truths are non-truths simply by exhibiting a counterexample – values for the variables that make the formula evaluate to false. But the only way to be sure that something is a truth is either to test it for all possible values of the variables or to prove it. Of course, there are an infinite number of possible values of the variables so testing is not generally effective unless we can limit it to a finite number of combinations by some sort of reasoning. Proving things is often the only way to be sure.

But even the activity of proving things requires an intuitive sense of what's true. There's no point in wasting your time trying to prove that $(a \times b) + x = (a + x) \times (b + x)$ when you can show it's not always true by letting a be 1, b be 2, and x be 3.

Furthermore, during proofs you will often be seeking some way to simplify the formula you're staring at and to do that you will need to know – or intuit – other truths.

Let me do an example from algebra. Let's prove that $(a+b)^2$ is $(a^2 + 2ab + b^2)$. Since we're being formal, we have to write this as $(a^2 + ((2 \times (a \times b)) + b^2))$. Let's assume the following truths:

`def-sq`:	$x^2 = x \times x$
`associativity-of-+`:	$((x + y) + z) = (x + (y + z))$
`commutativity-of-`$\times$:	$(x \times y) = (y \times x)$
$\times$`-2`:	$(2 \times x) = x + x$
`left-distributivity`:	$x \times (y + z) = (x \times y) + (x \times z)$
`right-distributivity`:	$(y + z) \times x = (y \times x) + (z \times x)$

[3] All of these truths implicitly assume the only objects in existence are the numbers or at least that the variables are understood to range only over numbers.

Theorem $(a+b)^2 = (a^2 + ((2 \times (a \times b)) + b^2))$

Proof:
$(a+b)^2$
$=$ {def-sq}
$(a+b) \times (a+b)$
$=$ {left-distributivity}
$((a+b) \times a) + ((a+b) \times b)$
$=$ {right-distributivity}
$((a \times a) + (b \times a)) + ((a+b) \times b)$
$=$ {right-distributivity}
$((a \times a) + (b \times a)) + ((a \times b) + (b \times b))$
$=$ {commutativity-of-×}
$((a \times a) + (a \times b)) + ((a \times b) + (b \times b))$
$=$ {associativity-of-+ }
$(a \times a) + ((a \times b) + ((a \times b) + (b \times b)))$
$=$ {associativity-of-+ }
$(a \times a) + (((a \times b) + (a \times b)) + (b \times b))$
$=$ {×-2}
$(a \times a) + ((2 \times (a \times b)) + (b \times b))$
$=$ {def-sq}
$a^2 + ((2 \times (a \times b)) + (b \times b))$
$=$ {def-sq}
$a^2 + ((2 \times (a \times b)) + b^2)$
□

This is a classic example of using previously established truths to derive a new truth. The basic strategy is to rewrite the formula (or, in this case, its left-hand side) repeatedly, using previously established truths. Each step above is an example of rewriting. We've discussed rewriting already but it is so basic it doesn't hurt to do it again. Recall that each rewrite step can be broken down into several mini-steps and you have to do all of them correctly to be confident of the result. Let's look at the mini-steps of just one rewrite step.

Consider the step justified by the second use of `associativity-of-+` above. The formula above the justification is transformed into the formula below it.

...
$(a \times a) + ((a \times b) + ((a \times b) + (b \times b)))$
$=$ {associativity-of-+ }
$(a \times a) + (((a \times b) + (a \times b)) + (b \times b))$
...

Here are the mini-steps involved in this single step.

general idea	*applied to our selected step*
(a) identify a target term to rewrite	we chose to focus on $((a \times b) + ((a \times b) + (b \times b)))$
(b) choose a previously proved truth to use; we'll assume for now that the truth is an equality	we chose `associativity-of-+`: $((x + y) + z) = (x + (y + z))$
(c) match the target with one side or the other of the chosen equality; choosing values for the variables in the equality truth	we matched our target with the right-hand side (rhs) of our equality truth; *target*: $(\underbrace{(a \times b)} + (\underbrace{(a \times b)} + \underbrace{(b \times b)}))$ *rhs*: $(\quad x \quad + (\quad y \quad + \quad z \quad))$ So let x be $(a \times b)$, let y be $(a \times b)$, and let z be $(b \times b)$. Since the equality is known to be true for all values of the variables, it is true for these values. Thus, we know this is true: $(((a \times b) + (a \times b)) + (b \times b))$ $=$ $((a \times b) + ((a \times b) + (b \times b)))$ and the right-hand side of this new equality is identical to our target
(d) replace the target by the new term we know it is equal to	we replaced the target by the left-hand side of the new equality, namely $(((a \times b) + (a \times b)) + (b \times b))$

These four mini-steps are repeated every time we rewrite. If the truth we're using is not just an equality but something more complicated, there are additional mini-steps. For example, if you want to use the truth that $(x \times (y/x)) = y$ when $x \neq 0$ to rewrite $(a \times (b/a))$ to b, you must be sure to prove that $a \neq 0$.

Since the whole purpose of mathematical logic is to make sure we reason without mistakes, it is necessary to describe the rules of inference precisely. I do that in the next chapter. But in this chapter I'm just foreshadowing the things we'll need.

But while I can explain the mini-steps in great detail, the real challenge in doing a proof is coming up with a broad strategy and finding the truths you need to carry it out. The broad strategy of the proof above is to "multiply-out $(a + b)^2$ and then collect like terms". To multiply a sum by a sum we'll need both left- and right-distributivity. To collect terms we generally need associativity and commutativity to rearrange the parentheses. We also need the definitions some basic operations, like how to square or double something. In algebra you have been taught these truths. But in our more general setting – where you are dealing

with unfamiliar functions with unfamiliar properties – you have to rely on your intuitions about what is true and what isn't. You have to simultaneously imagine a broad strategy, invent the necessary truths, use them to carry out your strategy, and prove the truths you need. Often, this involves some "search" on your part: you might find a strategy that works but discover that the necessary "truths" aren't true!

So it's important to develop your intutitions about what is always true and what isn't, and to develop your intuitions about how to find counterexamples.

By the way, the very same observations apply to debugging programs. You can demonstrate the existence of a bug with a suitable test. But you can't demonstrate the absence of a bug by testing. To *know* there are no bugs, you have to prove it.

• **QUESTION 131**: For each of the formulas below, decide whether it is always true or not. In ACL2, the variables range over all possible ACL2 objects. For each formula below that is *not* always true, show an ACL2 transcript in which the formula evaluates to false for chosen values of the variables. By the way, if you have a formula ψ with variables `p` and `q`, and want to test it for particular values of the variables, you could evaluate `(let ((p` α`) (q` β`))` ψ`)`, where α and β are expressions that produce the values you want to try for `p` and `q` respectively. You can list as many such variable bindings as you want. For example,

```
(let ((a 1) (b 2) (x 3))
  (equal (+ (* a b) x)
         (* (+ a x) (+ b x))))  ⟹ nil
```

Start your answer by clearly indicating the numbers of the formulas that are not always true. After that, paste the ACL2 transcript showing one false evaluation of each indicated formula. Do not turn in irrelevant evaluations.

```
 1.   (equal (not (not p)) p)
 2.   (iff (not (not p)) p)
 3.   (equal (and p q) (and q p))
 4.   (iff (and p q) (and q p))
 5.   (iff (and p (or q r)) (or p (and q r)))
 6.   (iff (and p (or q r)) (or (and p q) (and p r)))
 7.   (iff (or p (and a b)) (and (or p a) (or p b)))
 8.   (iff (not (and p q)) (or (not p) (not q)))
 9.   (iff (or (or p q) r) (or p (or q r)))
10.   (iff (implies p q) (or (and p q) (and (not p) (not q))))
11.   (iff (implies p q) (implies (not q) (not p)))
12.   (iff (implies p q) (or (not p) q)) •
```

• **QUESTION 132**: For each of the formulas below, decide whether it is always true or not. In ACL2, the variables range over all possible ACL2 objects. For each formula below that is *not* always true, show an ACL2 transcript in which the formula evaluates to false for chosen values of the variables. See the discussion in previous question about the use of `let` to run examples. You will need to add the definitions of `app`, `rev`, and `mem`, which are below. Start your answer by clearly indicating the numbers of the formulas that are not

always true. After that, paste the ACL2 transcript showing one false evaluation of each indicated formula. Do not turn in irrelevant evaluations.

1. `(equal (app nil x) x)`
2. `(equal (app x nil) x)`
3. `(equal (app (app x y) z) (app x (app y z)))`
4. `(equal (rev (rev x)) x)`
5. `(equal (rev (rev (rev x))) (rev x))`
6. `(equal (rev (app x y)) (app (rev x) (rev y)))`
7. `(not (equal (rev x) x))`
8. `(iff (mem e (app a b)) (or (mem e a) (mem e b)))`
9. `(implies (mem e (app a b)) (mem e a))`

Here are the relevant definitions.

```
(defun app (x y)
  (if (endp x)
      y
      (cons (first x)
            (app (rest x) y))))


(defun mem (x y)
  (if (endp y)
      nil
      (if (equal x (first y))
          t
          (mem x (rest y)))))


(defun rev (x)
  (if (endp x)
      nil
      (app (rev (rest x))
           (cons (first x) nil)))) •
```

3.7 Useful Identities (Assignment 8 cont'd)

Among the most useful identities are those relating the propositional functions, `and`, `or`, `not`, `implies`, and `iff`. You will use them over and over again. Until now, when you thought of "identities" in math you probably thought of equations. But the identities below are propositional identities, written with `iff`. They are typically used just like equalities: they tell you two expressions are propositionally equivalent and so you can replace one by the other in occurrences admitting propositional replacement.

Familiar Boolean Identities

identity	*name*
`(iff (not (not p)) p)`	Double Negation
`(iff (not (and p q))` `(or (not p) (not q)))`	De Morgan
`(iff (not (or p q))` `(and (not p) (not q)))`	De Morgan
`(iff (and p q) (and q p))`	Commutativity
`(iff (or p q) (or q p))`	Commutativity
`(iff (iff p q) (iff q p))`	Commutativity
`(iff (and (and p q) r)` `(and p (and q r)))`	Associativity
`(iff (or (or p q) r)` `(or p (or q r)))`	Associativity
`(iff (and p (or a b))` `(or (and p a) (and p b)))`	Distributivity
`(iff (or p (and a b))` `(and (or p a) (or p b)))`	Distributivity
`(iff (implies p q)` `(implies (not q) (not p)))`	Contraposition
`(iff (implies p q)` `(or (not p) q))`	Implicative Disjunction
`(iff (not (implies p q))` `(and p (not q)))`	Negated Implication

How do we know these identities hold? We could test them for all possible values of `p`, `q`, and `r`. Technically, there are an infinite number of possibile values for the variables. But every occurrence of every variable admits propositional replacement. Thus, rather than test them for the infinite number of true objects, we can limit our true test to the object `t`.

So we can establish these identities on all possible values of their variables by testing them on all combinations of `t` and `nil`.

A *tautology* is a propositional formula that evaluates to `t` for all combinations of Boolean values for its variables. All of the identities given above are tautologies.

• **QUESTION 133**: Below are two columns of expressions labeled A and B. For each expression in column A, find an expression in column B that is propositionally equivalent to it and note the familiar identity that establishes that equivalence, or else note that no expression in column B is equivalent. Just use a word processor to list the labels A1, ..., A9, and write down one of the B labels and an identity name beside each, or else write "none".

A1: `(not (not (endp a)))`

A2: `(or (not (mem e a))`

B1: `(and (endp x)`
`(not (equal x y)))`

```
        (mem e (app a b)))
A3:  (not (and (or a b) c))
A4:  (or (and x y) (and x z))
A5:  (not (or (endp x) (natp x)))
A6:  (implies (endp x)
              (not (natp x)))
A7:  (and (or a b) (and c d))
A8:  (not (implies (endp x)
                   (equal x y)))
A9:  (and (or (endp x) (and p q))
          (or (endp x) r))
```

```
B2:  (endp a)
B3:  (and (and (or a b) c) d)
B4:  (or (not (or a b)) (not c))
B5:  (implies (endp x) (natp x))
B6:  (or (mem e (app a b))
         (mem e a))
B7:  (and x (or y z))
B8:  (implies (mem e a)
              (mem e (app a b)))
B9:  (implies (not (not (natp x)))
              (not (endp x)))
B10:  (or (endp x)
          (and (and p q) r))
B11:  (or (and a b) (endp x))
```

•

• **Question 134**: This question is really a mini-project that will challenge the best of you. The problem is to define an ACL2 function that can decide whether a propositional formula is always true or not. A propositional formula that evaluates to true for all combinations of Boolean values for its variables is called a *tautology*. We will deal with tautologies at length later. It is probably obvious to you how propositional formulas can be represented as objects. For example, we can represent the *formula* `(implies (and p q) (or p q))` with the list `(implies (and p q) (or p q))`, i.e., the value of the expression `'(implies (and p q) (or p q))`. If you can imagine that, you can imagine a function `tautp` ("tautology predicate") that takes one input, an object representing a propositional formula, and returns `t` or `nil` to indicate whether the formula is a tautology. Define `tautp`. You may assume that the formula represented by the object is well-formed, composed only of variable symbols, `T` and `NIL`, and calls of the propositional functions `AND`, `OR`, `NOT`, `IMPLIES`, and `IFF`, that every function is applied to the right number of arguments and that `AND` and `OR` only take two arguments.

Hint: Implement a function, say `propositional-eval`, that takes an object representing a formula and a table binding variables to values, and returns the value of the formula. So, for example:

```
(propositional-eval '(implies (and p q) (or p q))
                    '((p t)
                      (q nil)))
```

evaluates to `t`. Also, implement the function `all-propositional-vars`, that takes a propositional formula and returns the list of all the variables in it, e.g.,

```
(all-propositional-vars '(implies (and p q) (or p q))) ⟹ (p q).
```

Then create a function that takes a list of variable symbols and returns the list of tables binding those variables to `t` and `nil` in all possible ways. Finally, `tautp` uses all of these functions to test each formula under each table. •

3.8 Summary (Assignment 8 cont'd)

In this chapter we saw the need for some basic truths, called "axioms," from which we derive others. We also saw a variety of "rules of inference," which are the tools we use to transform formulas. The ones we saw were

- rewriting, in which we use a previously proved truth to replace one term by another
- hypothesis, in which a hypothesis of the formula we're proving is used to replace one term by another
- cases, in which we consider an exhaustive set of possibilities and prove the formula in each case
- computation, in which we simply evaluate a term with no variables

There are two other rules we'll use. One is called "Tautology" and it just allows us to stop when the formula we're trying to prove is an instance of a previously proved truth. This is really just a special case of rewriting. The other is called "Constant Expansion" and allows us to express a constant in terms of function calls, e.g., `2` is `(+ 1 1)`.

One of the main complications we have to confront is dealing with formulas that are not just equalities. In particular, we need to deal with implications and propositional equivalences. To make our dealings with them simpler later, we build certain propositional identities directly into our terminology. For example, we will be able to treat

```
(implies (and p (and (not q) (not r))) s)
```

just as though it were

```
(implies (and p (and (not s) (not r))) q)
```

so that we can use the hypotheses `p`, `(not s)` and `(not r)` when working on `q`.

We describe our terminology and the rules in great detail in the next chapter.

4

Logic (Assignment 9: 7 days)

4.1 Preliminaries (Assignment 9 cont'd)

To state the rules of inference we need to speak precisely about terms.[1] In this section I define some notation and concepts that let us do that. We have already introduced several very important concepts for stating the rules of inference: the concept of an occurrence in a formula and the concept that an occurrence admits propositional replacement.

A *substitution* is a mathematical object that indicates how to replace certain variables by certain terms. The notation $\{v_1 \leftarrow \alpha_1, \ldots, v_k \leftarrow \alpha_k\}$ denotes a substitution that replaces the v_i by the α_i. In this notation, the v_i must be variables, they must all be distinct (no two the same), and the α_i must all be terms. We say the v_i are *hit* by the substitution. The elements of a substitution, i.e., each "$v_i \leftarrow \alpha_i$," is called a *binding* for the variable it hits, v_i. The *domain* of the substitution is the list of v_i; the *range* is the list of α_i.

Let σ be the substitution above. Then to *apply* σ to a term γ we simultaneously replace each occurrence of each v_i in γ by the corresponding α_i. Any variable not hit by the substitution is left unchanged. We denote by γ/σ the result of applying σ to γ. This is sometimes called *instantiating* γ with σ.

For example, let γ and σ be as shown below:

γ:
```
(g (f x y) x)
```

σ:
```
{ x ← (first a),
  y ← (rest a) ,
  a ← b }
```

Then

γ/σ:
```
(g (f (first a) (rest a)) (first a)).
```

There are two common mistakes students make when applying substitutions. The first is

[1] The ACL2 logic is much richer and more complex than described here. See [1].

to fail to replace every occurrence of every variable hit by the substitution. The following attempt to instantiate γ with σ is *wrong!*

```
(g (f (first a) (rest a)) x)
```

because we replaced the first occurrence of `x` but not the second. We have to replace every variable that is hit by the substititition.

The second common mistake is to replace each variable one at a time. For example, if we first replace `x` by `(first a)`, and then replace `y` by `(rest a)`, and then replace `a` by `b`, we get

```
(g (f (first b) (rest b)) (first b))
```

and that is *wrong!*. We must replace the variables *simultaneously*.

If there is a substitution σ such that γ/σ is γ', then we say γ' is an *instance* of γ. For example,

γ':
```
(iff (not (not (mem e (rev a))))
     (mem e (rev a)))
```

is an instance of γ, the Double Negation tautology,

γ:
```
(iff (not (not p)) p)
```

because γ' is γ/σ, where $\sigma = \{$`p` $\leftarrow$ `(mem e (rev a))`$\}$.

• **QUESTION 135**: What is γ/σ if:

γ: `(equal (app (app a b) c)`
`(app a (app b c)))`
σ: $\{$`a` $\leftarrow$ `(f b)`, `b` $\leftarrow$ `(f b)`, `c` $\leftarrow$ `(f b)`$\}$

You do not need ACL2 to answer this question; use a word processor. But be extra careful to type well formed, sensibly formatted text or else you'll get no credit! •

• **Question 136**: What is γ/σ if:

γ: `(cons 'a a)`
σ: $\{$`a` $\leftarrow$ `(f b)`$\}$ •

• **QUESTION 137**: Is there a substitution σ such that `(f x (h y z))`$/\sigma$ is `(f (g a) (h b 33))`? If so, what is it? If not, why? Just use a word processor; no ACL2. But make sure your answer is syntactically well-formed or you'll get no credit! •

• **Question 138**: Is there a substitution σ such that `(f x (h y y))`$/\sigma$ is `(f (g a) (h b`

`33))`? If so, what is it? If not, why? •

• **Question 139**: Is `(rev1 (rest x) (cons a b))` an instance of `(rev1 x a)`? •

• **Question 140**: Is `(rev1 (rest x) (cons a b))` an instance of `(rev1 x (rest a))`? •

• **Question 141**: `(mem (first a) (unn (int a b) (diff c d)))` is a σ instance of `(mem e (unn x y))`. What is σ? •

The *conjuncts list* of a term γ is a list of terms defined as follows. If γ is of the form `(and` p q`)`, its conjuncts list is the concatentation of the conjuncts list of p with the conjuncts list of q. The conjuncts list of the constant `t` (or `'t`) is the empty list. The conjuncts list of any other γ is the singleton list containing just γ.

For example, the conjuncts list of `(and a b c d)`, which you will recall is just an abbreviation for `(and a (and b (and c d)))`, is `(a b c d)`. The conjuncts list for `(and (and p t) (and q (or a b)))` is the list `(p q (or a b))`. The conjuncts list of `(equal x y)` is `((equal x y))`.

• **Question 142**: What is the conjuncts list of `(and (and a b) (and (or p q) d))`? •

Note: This concept is a little strange because it mixes ACL2 data objects (namely lists) with ACL2 syntactic objects (namely terms). Because terms look just like lists, this shouldn't cause you any trouble.

Given a list of terms, their *conjunction* (or the result of *conjoining* the terms in the list) is a term defined as follows. If the list is empty, its conjunction is the term `t`; if the list is a singleton containing α, the conjunction is α; otherwise, the conjunction is the term `(and` α β`)`, where α is the first term in the list and β is the conjunction of the rest of the list.

Thus, the conjunction of the list of the terms `(a b c)` is `(and a (and b c))`. I could alternatively say: the result of conjoining the list of terms `(a b c)` is `(and a (and b c))`.

Before going on with such definitions, let me illustrate how we'll use them. Suppose I said "to `and`-*right-associate* any expression, conjoin its conjunction list." Then you should be able to `and`-right-associate expressions:

expression	`and`-*right-associated version*
`(and (and a (and b c)) d)`	`(and a (and b (and c d)))`
`(and a b)`	`(and a b)`
`(mem e x)`	`(mem e x)`
`t`	`t`

Notice that an expression and its `and`-right associated version are propositionally equivalent, by the Associativity tautology for `and`. Our use of conjunct lists and conjoining is a way to implicitly "build in" to our reasoning applications of the Associative tautology.

Here is another typical use of this terminology. Suppose I wrote:

> Given a term other than `t`, obtain its conjuncts list, select one element, α, and let β be the result of conjoining the remaining elements.

Then you should be able to carry out this process. For example, if the term in question is `(and (and a b) (and c d))` and you select `c` for α, then β will be `(and a (and b d))`.

I now continue introducing new notation.

The *negation* of a term, α, is defined as follows. If α is of the form `(not` α'`)` its negation is α'. Otherwise, its negation is `(not` α`)`.

Thus, the negation of `(not (mem e a))` is `(mem e a)` and the negation of `(equal a b)` is `(not (equal a b))`. Obviously, the Double Negation identity is built into this syntactic concept.

The *pointwise negation* of a list of terms is the list of negations of each member of the list.

Thus, the pointwise negation of the list of terms `(a (not b) c)` is `((not a) b (not c))`.

The *disjuncts list* of a term γ is defined as follows. If γ is of the form `(or` p q`)`, its disjuncts list is the concatentation of the disjuncts list of p with the disjuncts list of q. If γ is of the form `(implies` p q`)`, its disjuncts list is the pointwise negation of the conjuncts list of p concatenated with the disjuncts list of q. Otherwise, the disjuncts list of γ is the singleton list containing γ.

For example, the disjuncts list of `(or (or a b) c)` is `(a b c)`.

The disjuncts list of

```
(implies (and (not p) q)
         (or (or a b) c))
```

is obtained by concatenating two parts:

- the pointwise negation of the conjuncts list of `(and (not p) q)`, which is `(p (not q))`, and
- the disjuncts list of `(or (or a b) c)`, which is `(a b c)`.

Concatenating these two produces the answer:

```
(p (not q) a b c).
```

Notice that `implies` is treated sort of like an `or`. In fact, we can use the propositional identities to justify this:

```
(implies (and (not p) q)
         (or (or a b) c))
```

is propositionally equivalent, by `Implicative Disjunction`, to

```
(or (not (and (not p) q))
    (or (or a b) c))
```

which is propositionally equivalent, by `De Morgan`, to

```
(or (or (not (not p)) (not q))
    (or (or a b) c))
```

which is propositionally equivalent, by `Double Negation`, to

```
(or (or p (not q))
    (or (or a b) c))
```

which is propositionally equivalent, by `Associativity`, to

```
(or p
    (or (not q)
        (or a
            (or b c)))),
```

and the disjuncts list of the last formula is `(p (not q) a b c)` as described. Thus, this definition of "disjuncts list" has several propositional identities built into it.

We will speak of two kinds of "factoring" of a formula. In one, we factor the formula into a "hypothesis" and a "conclusion." In the other, we factor it into a "pattern," a "replacement," and "maintained equivalence." In both cases, we build in various identities.

To *factor* formula ϕ into a *hypothesis* and a *conclusion* means to decompose the formula into two parts as follows. The decomposition isn't uniquely determined below – a formula can usually be factored more than one way. You are free to choose any of the options that apply to your ϕ. Let x be the disjunction list of ϕ. Then you may choose any element of x as the conclusion of your factoring of ϕ. Suppose you choose β as your conclusion. Then the hypothesis of this factoring of ϕ is the conjunction of the pointwise negation of the result of deleting β from x.

Wow! That's complicated! Ok, let's take it one step at a time. Let ϕ be `(implies (and (and a (not b)) c) d)`. Its disjunction list, called x above, is `((not a) b (not c) d)`. Now suppose we choose `d` for the conclusion of this factoring of ϕ. So β above is `d`. Now we have figure out the conjunction of the pointwise negation of the result of deleting β from x. The result of deleting β is `((not a) b (not c))`. If we take the pointwise negation of that, we get `(a (not b) c)`. And then if we take the conjunction of that, we get `(and a (and (not b) c))`. So that's the hypothesis of this factoring.

Thus, here is one way to factor ϕ:

ϕ:	`(implies (and (and a (not b)) c) d)`
hypothesis:	`(and a (and (not b) c))`
conclusion:	`d`

But if we choose `(not c)` as the conclusion of our factoring, then we get:

ϕ:	`(implies (and (and a (not b)) c) d)`
hypothesis:	`(and a (and (not b) (not d)))`
conclusion:	`(not c)`

Suppose we factor ϕ into a hypothesis ϕ_h and a conclusion ϕ_c. Then I claim that ϕ is

propositionally equivalent to (implies ϕ_h ϕ_c). Here's an explanation.

Start with ϕ

(implies (and p ...) q)

and use Implicative Disjunction to change it to

(or (not (and p ...)) q).

Then use De Morgan and Double Negation repeatedly to drive the not through all the conjuncts in the hypothesis and remove double negations (thus the "pointwise negation of the conjuncts") to get:

(or (or (not p) (not ...)) q).

Then, use Associativity of or to flatten it into a right-associated or-nest:

(or (not p) (or ... δ ... (or ... q)...)).

Now pick which of the disjuncts you want to use for your conclusion. Let's suppose it's δ above. By the Associativity and Distributivity of or we can move the δ to the last argument of the outermost or to get:

(or (or (not p) (or ... (or ... q)...)) δ).

Now use Implicative Disjunction again to turn that into

(implies (not (or (not p) (or ... (or ... q)...))) δ)

and then drive the not through the ors with De Morgan and use Double Negation to get

(implies (and p (and ... (and ... (not q))...)) δ).

Note that the hypothesis above, (and p (and ... (and ... (not q))...)), is the hypothesis created by factoring ϕ with conclusion δ. At every step we preserved propositional equivalence.

By defining the notion of factoring, we save ourselves all the work of those repeated little rewrites.

• **Question 143**: Can you factor (implies (and a b) c) so that the conclusion is (not c)? If so, show the hypothesis. If not, explain why not. •

• **Question 144**: Can you factor (implies (and (not a) b) c) so that the conclusion is (not a)? If so, show the hypothesis. If not, explain why not. •

• **Question 145**: Can you factor (implies (and (not a) b) c) so that the conclusion is a? If so, show the hypothesis. If not, explain why not. •

• **QUESTION 146**: Can you factor (implies (and (not a) b) (or c (not d))) so

that the conclusion is `c`? If so, show the hypothesis. If not, explain why not. •

• **Question 147**: Can you factor `(mem e (rev a))` so the conclusion is `(mem e (rev a))`? If so, show the hypothesis. If not, explain why not. •

A *refactoring* of a formula ϕ is any formula ϕ' that can be factored into the same hypothesis and conclusion as ϕ can be.

For example, a refactoring of `(implies (and a (and (not b) (not d))) (not c))` is `(implies (and (and a (not b)) c) d)`. Name these two formulas γ and γ'. We can factor γ into hypothesis `(and a (and (not b) (not d)))` and conclusion `(not c)`. But we can factor γ' into the same hypothesis and conclusion.

A formula ϕ may also be *factored* into three parts we call the *pattern*, the *replacement*, and the *maintained equivalence* as follows according to the form of ϕ. However, the options below are not uniquely determined by the form of ϕ. You are free to choose any of these options:

- if ϕ is of the form `(equal` x y`)`, the pattern is x, the replacement is y, and the maintained equivalence is `equal`.
- if ϕ is of the form `(iff` x y`)`, the pattern is x, the replacement is y, and the maintained equivalence is `iff`.
- if ϕ is of the form `(not` x`)`, the pattern is x, the replacement is `nil`, and the maintained equivalence is `equal`.
- regardless of the form of ϕ, the pattern is ϕ, the replacement is `t`, and the maintained equivalence is `iff`.

Furthermore, in all cases above, you may interchange the pattern and replacement.

Thus, if we say "factor" `(iff (consp x) (stringp y))` into pattern *pat*, replacement *rep* and maintained equivalence *eqv*, you might choose to let *pat* be `(consp x)`, *rep* be `(stringp y)`, and *eqv* be `iff`. But you could just as well let *pat* be `(stringp y)`, *rep* be `(consp x)`, and *eqv* be `iff`. Or, you could even let *pat* be `(iff (consp x) (stringp y))`, *rep* be `t`, and *eqv* be `iff`. You just have to decide how you want to factor it.

Here is how we'll use factoring and refactoring. How would you prove?

γ:
```
(implies (and (true-listp a)
              (equal b nil))
         (true-listp b))?
```

The formula reads "if `a` is a true list, and `b` is `nil`, then `b` is a true list." Intuitively, the proof would go something like "replace `b` in the conclusion by `nil`, creating `(true-listp nil)`, and then just evaluate it to `t`." Such an proof is justified by a rule of inference allowing us to "use" an equivalence hypothesis, by substituting one side for the other in the conclusion.

But now imagine you've been asked to prove

γ':
```
(implies (and (true-listp a)
              (not (true-listp b)))
         (not (equal b nil))).
```

Note that γ and γ' are propositionally equivalent. (This can be proved by repeatedly using the propositional identities Implicative Disjunction, De Morgan, Double Negation, Associativity, and Commutativity.) But in γ' we don't have an equality hypothesis. If the rule of inference for using an equivalence hypothesis is stated in the most straightforward way, it won't apply to γ' because there is no equality hypothesis in γ'. To solve this problem, our rule is stated very generally. It allows us to factor the formula, identify a hypothesis, substitute one side for the other, and then create any refactoring of the result. Thus, in one step, we can substitute `nil` for `b` to get:

```
(implies (and (true-listp a)
              (not (true-listp nil)))
         (not (equal b nil))).
```

This one step takes the place of the following sequence of steps:

```
(implies (and (true-listp a)
              (not (true-listp b)))
         (not (equal b nil)))
```

is propositionally equivalent, by Implicative Disjunction, De Morgan, Double Negation, Associativity, and Commutativity, to

```
(implies (and (true-listp a)
              (equal b nil))
         (true-listp b)),
```

which, by replacement of equals for equals, is equivalent to

```
(implies (and (true-listp a)
              (equal b nil))
         (true-listp nil)),
```

which, by Implicative Disjunction, De Morgan, Double Negation, Associativity, and Commutativity, is propositionally equivalent to

```
(implies (and (true-listp a)
              (not (true-listp nil)))
         (not (equal b nil))).
```

• **QUESTION 148**: Factor

```
(iff (or (findpos e a)
```

```
        (findpos e b))
    (findpos e (app a b)))
```

so the pattern is `(findpos e (app a b))`. What is the replacement? What is the maintained equivalence? •

• **Question 149**: Factor `(mem e (app a b))` so the pattern is `(mem e (app a b))`. What is the replacement? What is the maintained equivalence? •

4.2 Syntax (Assignment 9 cont'd)

Now for a shocking revelation: most computer scientists do not use ACL2 notation! We've stuck to ACL2 notation so far because it makes it easier (I think) for you to understand such ideas as how to do substitution, address a subterm, factor a formula into a certain forms, etc. ACL2 also takes you out of your "comfort" zone with arithmetic and lets us prove simple but non-obvious theorems about lists. But now it is time to learn the traditional notation.

Most computer scientists write in "infix" notation, writing $x+y$ instead of `(+` x y`)`. You're familiar with infix notation for addition (+), subtraction (−), multiplication (×), and division (/), as well as equality (=) and dis-equality (≠) and the standard inequalities (<, ≤, ≥, and >). You are free to use them on your homeworks. You may choose to type your answers, which raises questions like "how do I type special symbols like ≤ and →?" I'll give you a table of stubstitutes below. But please understand that the ACL2 system does not accept infix!

The familiar notation, e.g., "$x+y$", is an example of what computer scientists call *concrete syntax*. It is the syntax of the "source language." The ACL2 notation, e.g., "`(+ x y)`", is an example of *abstract syntax*. It is the deep, underlying syntactic structure of the utterance. Typically, the concrete syntax provides all sorts of conveniences while the abstract syntax is very regular and uniform. For example, in traditional concrete syntax one might write "$x+y$", "xy," "x^y", "$mod(x,y)$", and "$\sqrt[x]{y}$" where the corresponding abstract syntactic structures would be "`(+ x y)`", "`(* x y)`", "`(expt x y)`", "`(mod x y)`", and "`(root x y)`". Note that the concrete syntax varies wildly: the operator (the function symbol) is written between the operands, the operator is not written at all (!), conveyed by raising or lowering the symbols, written before the operands, or written as a squiggly mark around the operands depending on what the operator is. But regardless of the concrete syntax, the abstract syntax is very regular: a function symbol applied to two arguments.

Even though ACL2 doesn't accept infix notation, I will use it interchangeably with ACL2 notation in this course. I may well write, $(x + (6 \times y)) \neq j$ to mean `(not (equal (+ x (* 6 y)) j))`. Note how "≠" turns into an `equal` expression embedded in a `not`.

We will *always* give our terms the meaning that ACL2 gives them. So you should not be surprised if I say `nil` $+ 6 = 6$. We hope such weird expressions don't arise! But if they do, don't argue with me about what they mean! Infix notation is just shorthand for ACL2 and

infix expressions mean whatever ACL2 says their ACL2 counterparts mean!

Computer scientists use a lot more symbols than the arithmetic ones you've seen before. We will also start to use those symbols and infix notation. Sometimes people will use other symbols for these connectives and we show common alternatives below. If someone uses a symbol you don't recognize, ask them what it means! In this booklet, we'll either write in ACL2 or we will use the *infix* column of symbols and write in infix.

<table>
<tr><th>ACL2</th><th>infix</th><th>ASCII</th><th>other alternatives</th></tr>
<tr><td><code>t</code></td><td>$True$</td><td></td><td>1</td></tr>
<tr><td><code>nil</code></td><td>$False$</td><td></td><td>0</td></tr>
<tr><td><code>(equal x y)</code></td><td>$x = y$</td><td><code>x = y</code></td><td></td></tr>
<tr><td><code>(not (equal x y))</code></td><td>$x \neq y$</td><td><code>x != y</code></td><td></td></tr>
<tr><td><code>(and p q)</code></td><td>$(p \wedge q)$</td><td><code>(p && q)</code></td><td><code>(p * q)</code>
<code>(p & q)</code></td></tr>
<tr><td><code>(or p q)</code></td><td>$(p \vee q)$</td><td><code>(p || q)</code></td><td><code>(p | q)</code></td></tr>
<tr><td><code>(not p)</code></td><td>$(\neg p)$</td><td><code>(~ p)</code></td><td>$\overline{p}$
<code>(- p)</code>
<code>(! p)</code></td></tr>
<tr><td><code>(implies p q)</code></td><td>$(p \rightarrow q)$</td><td><code>(p --> q)</code></td><td>$(p \supset q)$
$(p \Rightarrow q)$
<code>(p ==> q)</code></td></tr>
<tr><td><code>(iff p q)</code></td><td>$(p \leftrightarrow q)$</td><td><code>(p <---> q)</code></td><td>$(p \equiv q)$</td></tr>
</table>

You may mix infix notation for `equal` ("`=`"), `and` ("`&&`") `or` ("`||`"), `not` ("~" aka "tilde"), `implies` ("`-->`"), `iff` ("`<--->`") and arithmetic (`+`, `-`, `*`, `<`, `>`, `<=` and `>=`). (The last two are the ASCII substitutes for "≤" and "≥". Use normal ACL2 notation for all other function calls.

Substitutions, such as

{ `x` ← `(first x)`, `y` ← `(second x)` }

should be displayed in ASCII as

```
{ x <-- (first x), y <-- (second x) }.
```

Do not use alternative notation, e.g., don't write 0 for false! Write or spell out Greek letters. Thus, if you want to write "$(\alpha \wedge \beta) \rightarrow (\gamma \vee \delta)$" and you're entering the formula at a keyboard, type "`(alpha && beta) --> (gamma || delta)`" and avoid using Greek letter names as variables!

Use parentheses to disambiguate. For example, instead of writing "`~A || B`" write either "`(~ A) || B`" or "`~ (A || B)`," depending on which you mean. You may write "`A && B && C`" instead of "`A && (B && C)`" or "`(A && B) && C`" since **and** is associative. The same flexibility is allowed for "`||`", "`+`," and "`*`," since they are also associative. Otherwise, make sure that you provide the right number of arguments to every function whether you're writing in infix or ACL2.

When writing infix you may drop the outermost pair of parentheses. Thus, instead of writing

```
((x = y) → ((first x) = (first y)))
```

you may write

```
(x = y) → ((first x) = (first y)).
```

But if the formula is not at the top level, you should write the parentheses around it. For example, if I wanted to conjoin the formula above with `(natp x)` I would have to write:

```
(natp x) ∧ ((x = y) → ((first x) = (first y))).
```

not

```
(natp x) ∧ (x = y) → ((first x) = (first y))
```

since the latter could ambiguously also mean ((natp x) ∧ (x = y)) → ((first x) = (first y)).

Recall the earlier lesson on pretty printing. It is simply impossible to read stuff like this:

```
(defun insert (e x) (if (endp x) (cons e x) (if (lexorder e (first x)) (cons e x
) (cons (first x) (insert e (rest x))))))
```

For what it is worth, here is that text along-side an 80 column "ruler"

```
         10        20        30        40        50        60        70        80
123456789|123456789|123456789|123456789|123456789|123456789|123456789|123456789|
(defun insert (e x) (if (endp x) (cons e x) (if (lexorder e (first x)) (cons e x
) (cons (first x) (insert e (rest x))))))
```

Learn to "pretty print" your text so that the format of the text helps convey the structure of the formula! I would display the definition above this way:

```
(defun insert (e x)
   (if (endp x)
```

```
      (cons e x)
      (if (lexorder e (first x))
          (cons e x)
          (cons (first x)
                (insert e (rest x))))))
```

This advice will help you avoid mistakes! The point is not to squeeze the text into the space allowed. The point is to record and communicate your ideas. Remember: You read your stuff more often than anybody else does and if you express yourself clearly you'll understand your own thinking better. Habits developed now will help you throughout your career.

I will repeat the three heuristic guidelines given earlier. (i) Look for ways to break up a line if it contains more than about 25 non-blank characters and try never to produce a line with more than about 40 characters. (ii) If you have to break up any argument, first indent that argument and all of its peers. (iii) Indent every argument equally.[2]

For example, while I would write this:

```
(if (endp y) nil (equal x (first y)))
```

because the whole `if`-expression fits in less than 40 characters. But if I had to write something slightly longer, like

```
(if (endp y) nil (cons (first x) (app (cdr x) y)))
```

I would break it up. I would not do this:

```
(if (endp y) nil (cons (first x)
                       (app (cdr x) y)))
```

because it violates guideline (ii). Instead, I would indent the `cons`-expression and indent the `nil` equally:

```
(if (endp y)
    nil
    (cons (first x) (app (cdr x) y)))
```

If I felt the `cons`-expression was still too hard to read, I would indent its arguments:

```
(if (endp y)
    nil
    (cons (first x)
          (app (cdr x) y)))
```

The point is *not* just to make the text fit in 80 columns! The point is to make the structure of the formula clear. The indentation tells me which expressions belong to which function call.

[2] `Defun` is an exception: it is conventional to write the "`defun`", function name and list of formals on one line and then indent the body a space or two.

When you use infix, follow similar guidelines. In particular, don't type stuff like this:

```
((true-listp a) && (true-listp b)) --> (first (if (endp a) b (cons (first a) (ap
p (rest a) b)))) = (if (endp a) (first b) (first a))
```

If it's hard for you to read, you'll make mistakes. In any case, the graders and I won't read it.

Instead, pretty print your infix too:

```
 (  (true-listp a)
  &&
    (true-listp b))
-->
 (  (first (if (endp a)
               b
               (cons (first a)
                     (app (rest a) b))))
   =
     (if (endp a) (first b) (first a))).
```

Notice how all the arguments to the "&&" and to the "=" are indented, they are indented equally, and they are indented more than the "-->" so, again, the indentation helps the reader understand the structure.

Some people prefer to indent the infix operators and their arguments equally. That's ok too.

```
 ((true-listp a)
  &&
  (true-listp b))
-->
  ((first (if (endp a)
              b
              (cons (first a)
                    (app (rest a) b))))
   =
   (if (endp a) (first b) (first a))).
```

Sometimes when you use infix to write a proof, you try to line up the terms on adjacent lines of the proof to help show how a line is derived from the previous one:

```
  ((~(endp x)) && (perm (rest x) (rest x))) --> (perm x x)
<--> {perm, hyp, if-nil}
  ((~(endp x)) && (perm (rest x) (rest x))) --> ((mem (first x) x)
                                                 &&
                                                 (perm (rest x)
```

```
                                                                  (del1 (first x) x)))
```

The display above clearly shows that the hypotheses of the formula remained the same and I only changed the conclusion. It violates the guideline of not breaking up a subformula if you print its peers on a single line. But the overarching consideration has to be clarity of the totality, not strict rules for how to print (other than avoiding lines extending past the 80th column). So I would tend to display the transformation as shown above, which just fits in 80 columns. However, since the `(perm (rest x) (del1 (first x) x))` is now close to the right margin, I would be in trouble if I had to keep expanding it. So if I need more space for subsequent transformations, I'd redisplay the formula to gain some room on the right. I would note explicitly that I was doing so, so that the display above would become:

```
  ((~(endp x)) && (perm (rest x) (rest x))) --> (perm x x)
<--> {perm, hyp, if-nil}
  ((~(endp x)) && (perm (rest x) (rest x))) --> ((mem (first x) x)
                                                  &&
                                                  (perm (rest x)
                                                        (del1 (first x) x)))
<--> {redisplaying}
  ((~(endp x))
   &&
   (perm (rest x) (rest x)))
 -->
  ((mem (first x) x)
   &&
   (perm (rest x)
         (del1 (first x) x)))
```

If you learn nothing else from this course than the value of communicating complicated ideas clearly you will benefit!

When in doubt about how to display a formula, talk to me or your TA. But the bottom line is this: use every opportunity to help the reader understand your ideas. This advice applies as much to your mathematical ideas as your programming and your software design.

Warning: You will get a 0 for any answer that violates the following rules. Every "formula" must be syntactically well-formed. Nothing should extend beyond the 80th column – lines should not be truncated or wrap around when viewed in a window 80 characters wide. Finally, if the grader feels that insufficient effort has been made to display a formula sensibly, he or she is free to assign a 0 for that answer even if it is otherwise correct!

In the rest of these notes, we may well write the propositional formula

```
(implies (and p (or q (not r)))
         (iff u v))
```

as $(p \wedge (q \vee (\neg r))) \rightarrow (u \leftrightarrow v)$. I try to follow the same rules I gave you above.

You should just get used to the idea that "$(p \wedge q) \rightarrow r$" is an often-used concrete syntax for the abstract syntax "`(implies (and p q) r)`." I think of the former as merely a way to display the latter.

The *truth values* (more commonly called *Booleans*) are $True$ and $False$. In these notes when I write infix formulas, I'll use $True$ and $False$ as synonymous with `t` and `nil`, respectively.[3]

ACL2 is an unconventional logic due to its treatment of the Booleans: in ACL2 Booleans are treated like any other objects.

In conventional logics, the truth values are given special treatment. Truth values are not among the "conventional" objects like numbers and conses. The logical operators ("$\wedge$", "$\vee$", "$\neg$", "$\rightarrow$", and "$\leftrightarrow$") are applied exclusively to (expressions yielding) truth values. Predicates and relations, like equality ("$=$") and less than ("$<$") operate exclusively on the conventional objects but return truth values. Functions, like "$+$" and `cons`, operate exclusively on conventional objects and return conventional objects. Thus, in a conventional logic you would never see anyone write "$(\mathtt{x} < 23) = (\mathtt{y} = \mathtt{z})$".

But such formulas have meaning in ACL2. The formula above, translated to ACL2, `(equal (< 23) (equal y z))`, means that the values of "$(\mathtt{x} < 23)$" and "$(\mathtt{y} = \mathtt{z})$" are identical. The fact that the values in question happen to be "truth values" is unimportant in ACL2. One reason ACL2 (and Lisp) allow this is so that we can write logical operators inside the bodies of "conventional" function definitions.

But out of respect for conventions, in this book I will avoid using these features of ACL2 except to make certain important points (about the role of "congruences" in reasoning). When I use the features, I'll highlight the fact that I'm doing so.

The following truth is very useful in this regard:

```
(implies (and (booleanp p)
              (booleanp q))
         (iff (equal p q)
              (iff p q)))
```

or, in our new notation

```
((booleanp p) ∧ (booleanp q)) → ((p = q) ↔ (p ↔ q)).
```

Recall the unconventional expression $(\mathtt{x} < 23) = (\mathtt{y} = \mathtt{z})$. This can be written conventionally as $(\mathtt{x} < 23) \leftrightarrow (\mathtt{y} = \mathtt{z})$, since both sides of the "$\leftrightarrow$" are Boolean.

• **Question 150**: Write the infix form of the propositional formula.

```
(iff (not (and p q))
     (or (not p) (not q))) •
```

• **QUESTION 151**: Write the infix form of the propositional formula.

[3] In ACL2, `t` is a different symbol from `True`. But note the font difference between ACL2's symbol `True` and $True$, my abbreviation for the symbol `t` in infix formulas.

```
(implies (and (not c)
              (implies (implies b (not c)) a))
         (implies (implies a c) nil))
```
•

• **Question 152**: Write the ACL2 form of the following propositional formula.

$$((p \to q) \wedge (q \to r)) \to (p \to r) \;\; \bullet$$

• **Question 153**: Write the ACL2 form of the following propositional formula.

$$((B \to (\neg C)) \to A) \leftrightarrow (((\neg A) \to B) \wedge ((\neg A) \to C)) \;\; \bullet$$

• **Question 154**: Write the ACL2 form of $(p \vee (\neg p)) \leftrightarrow True$. •

• **Question 155**: What is the conjuncts list of $a \wedge ((b \vee c) \wedge (\neg d))$? •

• **Question 156**: Is $(a \wedge (b \vee c)) \to (a \to b)$ a σ instance of $(p \wedge q) \to r$? If so, what is σ? If not, why? •

• **Question 157**: Let σ be the substitution $\{x \leftarrow (a+b), y \leftarrow c, z \leftarrow (i \times j)\}$. Let α be $x + 2 \times y \leq 25 \times z$. What is α/σ? •

• **Question 158**: If σ is the substitution $\{p \leftarrow$ `(ENDP A)`$, q \leftarrow$ `(ENDP (APPEND X Y))`$\}$ what is $(p \to q)/\sigma$? •

• **Question 159**: Factor $(a \wedge b) \to c$ so that the conclusion is $(\neg a)$. What is the hypothesis? •

• **QUESTION 160**: Factor $((a \wedge b) \wedge (\neg(p \leftrightarrow q))) \to c$ so the conclusion is $(p \leftrightarrow q)$. What is the hypothesis? •

• **Question 161**: Factor `(rev (append a b))` = `(append (rev b) (rev a))` so that the pattern is `(append (rev b) (rev a))`. What is the replacement? What is the maintained relation? •

• **Question 162**: Factor `(ordp (isort x))` so that the pattern is `(ordp (isort x))`. What is the replacement? What is the maintained equivalence? •

• **Question 163**: Factor `(evenp (+ x y))` $\leftrightarrow$ (`(evenp x)` $\wedge$ `(evenp y)`) so that the pattern is `(evenp (+ x y))`. What is the replacement? What is the maintained equivalence? •

• **QUESTION 164**: Factor `(`$\neg$ `(mem e x))` so the pattern is `(mem e x)`. What is the replacement? What is the maintained equivalence? •

• **Question 165**: Factor `(rev (rev x))` = `x` so that the pattern is `(rev (rev x))` = `x`. What is the replacement? What is the maintained equivalence? •

Technically, in the next section, we ought to include axioms for the symbol = (`equal`). But we assume you know about equality. The key facts about equality (that aren't written elsewhere here) are:

- The function symbol `equal` is Boolean, it returns `t` or `nil`.
- Different constants are unequal: `t` $\neq$ `nil`, `0` $\neq$ `1`, `17` $\neq$ `-23`, etc.

Those familiar with axiomatizations of equality might wonder about the Symmetry and Transitivity of Equality. Those can be proved using our rules of inference. So can the usual infinity of axioms about functional reflexivity, e.g., $((x_1 = y_1) \wedge (x_2 = y_2)) \rightarrow$ `(+` x_1 x_2`)` $=$ `(+` y_1 y_2`)`.

4.3 Axioms (Assignment 10: 2 days)

In this section I present the axioms we'll use.[4]

Recall that "axioms" are our "ground truths," formulas that we accept as truths: true for all values of the variables in them.

You can think of the axioms of a system as describing the properties that the symbols have. For example, you know that if you build a cons with `(cons x y)` and then take its `first`, you'll get `x`, no matter what values for `x` and `y` you use. That gives rise to the axiom named `first-cons` below.

Axiom `equal-reflexive`:
`x = x`

Axiom `equal-boolean`:
`((x = y) = t) ∨ ((x = y) = nil).`

Axiom `if-ax1`:
`(x ≠ nil) → ((if x y z) = y).`

Axiom `if-ax2`:
`(x = nil) → ((if x y z) = z).`

Axiom `first-cons`:
`(first (cons x y)) = x.`

Axiom `rest-cons`:
`(rest (cons x y)) = y.`

Axiom `cons-first-rest`:
`((consp x) = t) → ((cons (first x) (rest x)) = x).`

Axiom `consp-cons`:
`(consp (cons x y)) = t.`

Axiom `first-atom`:
`(¬(consp x)) → ((first x) = nil).`

[4] I omit many axioms of the ACL2 logic – about rationals, complex rationals, characters, strings, and symbols. I just don't think we'll need them. If we do, I'll add them!

Axiom `rest-atom`:
`(¬(consp x)) → ((rest x) = nil)`.

The recognizers are all Boolean.

Axiom `natp-boolean`:
`((natp x) = t) ∨ ((natp x) = nil)`.

Axiom `stringp-boolean`:
`((stringp x) = t) ∨ ((stringp x) = nil)`.

Axiom `symbolp-boolean`:
`((symbolp x) = t) ∨ ((symbolp x) = nil)`.

Axiom `consp-boolean`:
`((consp x) = t) ∨ ((consp x) = nil)`.

The naturals, strings, symbols, and conses are disjoint. We refer to all of all these axioms by the single name "`disjointness`."

Axiom `disjointness`:
`((consp x) = t) → ((natp x) = nil)`.

Axiom `disjointness`:
`((consp x) = t) → ((stringp x) = nil)`.

Axiom `disjointness`:
`((consp x) = t) → ((symbolp x) = nil)`.

Axiom `disjointness`:
`((natp x) = t) → ((stringp x) = nil)`.

Axiom `disjointness`:
`((natp x) = t) → ((symbolp x) = nil)`.

Axiom `disjointness`:
`((stringp x) = t) → ((symbolp x) = nil)`.

In addition, we assume an infinite number of axioms about the "types" of our constants. We refer to all these axioms by a single name, `type`.

Axiom Schema `type`:
`(natp` *n*`) = t`
for every natural number object *n*.

Axiom Schema `type`:
`(stringp` *str*`) = t`
for every string object *str*.

Axiom Schema `type`:
`(symbolp '`*sym*`) = t`
for every symbol object *sym*.

Axiom Schema `type`:
`(consp '`*x*`) = t`

for every cons object x.

Finally, every `defun` introduces an axiom.

Definitional Axiom Schema: Any admissible definition of the form

`(defun` f `(`v_1 ... v_n`)` β`)`

introduces the definitional axiom:
Def Ax f: `(`f v_1 ... v_n`)` = β.

Recall that the definitional principle has four conditions for admissibility. These conditions insure that the logic stays sound: only valid formulas can be proved. But we won't discuss this.

We may add additional axioms later.

4.4 Rules of Inference (Assignment 10 cont'd)

I wrote this section succinctly – no cluttering examples – so you could come back to it again and again as a reference manual. I recommend you put a bookmark here and refer back every time we use an inference rule until you know them cold! In Section 4.6 I work through many examples.

Each Rule of Inference tells that you can replace a goal, ψ, by a new goal, ψ', while preserving propositional equivalence. Thus, if a rule is applied correctly, $\psi \leftrightarrow \psi'$ is true. We will see that "$\leftrightarrow$" (`iff`) is transitive. So if you can get a chain of replacements:

ψ
$\leftrightarrow$ $\{Rule_1\}$
ψ_1
$\leftrightarrow$ $\{Rule_2\}$
... ...
$\leftrightarrow$ $\{Rule_k\}$
ψ_k

in which each step is justified by one of the Rules of Inference below, then you have shown that formula $\psi \leftrightarrow \psi_k$. If ψ_k is $True$, then you've proved ψ.

That is, a proof of ψ is a proof of $\psi \leftrightarrow True$, i.e., a proof of $\psi \leftrightarrow$ `t`.

A *proof* of ψ is a reduction of ψ to $True$ by a finite number of applications of the Rules of Inference below.

The same proof may be described in many styles. I do not restrict you to any particular style.

A *Theorem* is any axiom, definitional axiom, or formula that can be proved. A *Tautology* is a propositional theorem, i.e., a formula that only involves the propositional connectives

and variables and that can be proved. Sometimes theorems are called by other names, including *Lemma* (a theorem proved in order to use it as a stepping-stone in another proof) and *Corollary* (a theorem whose proof follows easily from a theorem just proved).

There are six rules: Tautology (or Theorem), Rewrite, Hypothesis, Cases, Constant Expansion, and Computation. Tautology (or Theorem) and Computation are just special cases of Rewrite.

Rule of Inference Tautology (or Theorem) ("Taut" or "Thm"):

Summary: This rule allows you to prove your goal ψ if it is an instance of a theorem ϕ. When ϕ is actually a tautology, we call this the Tautology rule. Otherwise, it is the Theorem rule.

Description: If you can carry out all of the following steps:

1. find a theorem ϕ
2. find a substitution σ such that goal ψ is ϕ/σ

then $\psi \leftrightarrow True$.

End of Rule

Rule of Inference Rewrite ("Re"):

Summary: This rule allows you to rewrite part of the goal ψ using a previously established theorem (including an axiom or definition) ϕ.

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c so that the term you wish to replace is in the goal's conclusion ψ_c
2. let π be the address in ψ_c pointing to the subterm you want to replace
3. factor the previously proved theorem ϕ into a hypothesis ϕ_h and conclusion ϕ_c
4. factor ϕ_c into pattern α, replacement β, and maintained equivalvence *eqv* such that the term you want to replace (at π in ψ_c) is an instance of α under some substitution σ
5. if *eqv* is `iff`, confirm that address π admits a propositional replacement in ψ_c
6. prove $\psi_h \rightarrow (\phi_h/\sigma)$
7. obtain ψ'_c by replacing the term at π in ψ_c by β/σ

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow \psi'_c$.

End of Rule

Note: Step 6 is called *relieving the hypotheses* of the theorem ϕ. *Forgetting or misunderstanding this step is probably the most common mistake students make.* Using an implication,

like $\phi_h \rightarrow (\alpha = \beta)$ or $\phi_h \rightarrow (\alpha \leftrightarrow \beta)$, to rewrite with is called *conditional rewriting*. Most rewriting is conditional.

The Relieve-Hyps Waiver: If the formula of step 6 is a tautology or can be proved merely by rewriting with axioms and definitions, evaluation (see below), and tautologies, you need not mention its proof. Otherwise, describe how that formula is proved when you cite the use of the Rewrite rule of inference.

Please Note: Recall the mini-steps of rewriting (page 85 and 102). If you compare the informal discussion of the mini-steps to the precise one you should be able to see the correspondence, except that the precise description must make allowances for using propositional equivalence to do replacement and must deal with relieving the hypotheses of conditional rewrites. These step-by-step rules are incredibly detailed and precise. Use the examples in Section 4.6 to debug your understanding. Practice until you can carry out each rule correctly. I am not interested in the individual steps involved in using a rule, just in whether the end result is exactly as described here.

Rule of Inference Hypothesis (or "Hyp"):

Summary: This rule allows you to use one of the hypotheses of the goal ψ by replacing some term in its conclusion with an equivalent term.

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c
2. let δ be one of the conjuncts of ψ_h (a hypothesis of goal ψ)
3. factor hypothesis δ into pattern α, replacement β, and maintained equivalence *eqv*
4. let π be an address in ψ_c at which you find α
5. if *eqv* is `iff`, confirm that address π admits a propositional replacement in ψ_c
6. obtain ψ'_c by replacing the term at π in ψ_c by β

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow \psi'_c$.

Warning: Unlike Rewrite, the Hypothesis rule does not allow you to instantiate α! There is no substitution σ in this rule!

End of Rule

Rule of Inference Cases ("Cases"):

Summary: This rule allows you to prove ψ by splitting the proof into several cases. The case-split is justified by some theorem ϕ which establishes that all possibilities have been considered.

Description: If you can carry out all of the following steps:

1. choose some theorem ϕ
2. let σ be a substitution that instantiates the variables of theorem ϕ
3. let $\phi'_1, \phi'_2, \ldots \phi'_k$ be the disjuncts of ϕ/σ
4. prove $\phi'_i \rightarrow \psi$, for each $1 \leq i \leq k$

then $\psi \leftrightarrow True$.

End of Rule

Rule of Inference Constant Expansion ("Const"):

Summary: This rule allows you to replace certain constants by function calls, or vice versa. There are several variants.

Description of Variant 1: If you can carry out both of the following steps:

1. let π be a legal address in ψ that points to a non-empty list constant c
2. let ψ' be the result of replacing the term at address π in ψ by `(cons '`c_1 `'`c_2`)`, where c_1 is the first element of c and c_2 is the rest of c

then $\psi \leftrightarrow \psi'$.

Description of Variant 2: If you can carry out both of the following steps:

1. let π be a legal address in ψ that points to constant term `'`n, where n is a positive natural number
2. let ψ' be the result of replacing the term at address π in ψ by `(+ 1 '`n'`)`, where n' is $n - 1$

then $\psi \leftrightarrow \psi'$.

End of Rule

Note: Thus, the constant `'(M W F)` may be replaced by `(cons 'M '(W F))`. The two forms above essentially tell us how to build all list constants and naturals by the primitive operations `cons` and `+`. We omit forms of this rule that reveal the structures of strings, symbols, characters, or non-natural numbers because we won't use them.

Rule of Inference Computation ("Comp"):

Summary: This rule allows you to replace certain function calls by constants.

Description: If you can carry out all of the following steps:

1. let π be a legal address in ψ that points to a function call `(`f $c_1 \ldots c_n$`)` where all the c_i are constant terms

2. the function f is defined and it is possible to compute the value, c, returned by `(`$f\ \ c_1 \ldots c_n$`)` from axioms and definitions by repeated use of Rewrite and Constant Expansion and the definitional axioms

3. c is a constant

4. ψ' is obtained from ψ by replacing the term at π in ψ by `'`c

Then $\psi \leftrightarrow \psi'$.

End of Rule

We will add other rules of inference later.

4.5 Tautologies (Assignment 10 cont'd)

4.5.1 Familiar Tautologies

You *must* be able to recognize many common tautologies at a glance! You will note that the first fourteen below are the previoiusly mentioned named identities (page 106), with a more commonly used variant of Contraposition included.

$(\neg(\neg p)) \leftrightarrow p$	Double Negation
$(\neg(p \wedge q)) \leftrightarrow ((\neg p) \vee (\neg q))$	De Morgan
$(\neg(p \vee q)) \leftrightarrow ((\neg p) \wedge (\neg q))$	De Morgan
$(p \wedge q) \leftrightarrow (q \wedge p)$	Commutativity
$(p \vee q) \leftrightarrow (q \vee p)$	Commutativity
$(p \leftrightarrow q) \leftrightarrow (q \leftrightarrow p)$	Commutativity
$((p \wedge q) \wedge r) \leftrightarrow (p \wedge (q \wedge r))$	Associativity
$((p \vee q) \vee r) \leftrightarrow (p \vee (q \vee r))$	Associativity
$(p \wedge (q \vee r)) \leftrightarrow ((p \wedge q) \vee (p \wedge r))$	Distributivity
$(p \vee (q \wedge r)) \leftrightarrow ((p \vee q) \wedge (p \vee r))$	Distributivity
$(p \rightarrow q) \leftrightarrow ((\neg q) \rightarrow (\neg p))$	Contraposition
$((p \wedge r) \rightarrow q) \leftrightarrow (((\neg q) \wedge r) \rightarrow (\neg p))$	Contraposition
$(p \rightarrow q) \leftrightarrow ((\neg p) \vee q)$	Implicative Disjunction
$(\neg(p \rightarrow q)) \leftrightarrow (p \wedge (\neg q))$	Negated Implication
$p \vee (\neg p)$	Basic
$p \rightarrow p$	Basic
$p \leftrightarrow p$	Basic
$(p \wedge q) \rightarrow p$	Basic
$(False \wedge p) \leftrightarrow False$	Short-Circuit
$(False \vee p) \leftrightarrow p$	Short-Circuit
$(True \wedge p) \leftrightarrow p$	Short-Circuit
$(True \vee p) \leftrightarrow True$	Short-Circuit
$(True \rightarrow p) \leftrightarrow p$	Short-Circuit
$(False \rightarrow p) \leftrightarrow True$	Short-Circuit
$(p \rightarrow True) \leftrightarrow True$	Short-Circuit
$(p \rightarrow False) \leftrightarrow (\neg p)$	Short-Circuit
$(p \leftrightarrow q) \leftrightarrow ((p \rightarrow q) \wedge (q \rightarrow p))$	Def $\leftrightarrow$
$(p \wedge p) \leftrightarrow p$	Contraction
$(p \vee p) \leftrightarrow p$	Contraction
$(p \wedge (p \rightarrow q)) \leftrightarrow (p \wedge q)$	Forward Chaining
$(p \wedge (p \rightarrow q)) \leftrightarrow (p \wedge (p \rightarrow q) \wedge q)$	Forward Chaining
$((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$	Transitivity
$((p \leftrightarrow q) \wedge (q \leftrightarrow r)) \rightarrow (p \leftrightarrow r)$	Transitivity
$(p \rightarrow (q \rightarrow r)) \leftrightarrow ((p \wedge q) \rightarrow r)$	Promotion

These formulas should be *second nature* to you as a computer scientist! You should recognize them not because you've memorized them but because they are *obvious* truths to you because you understand what the symbols mean.

4.5.2 Truth Tables

You will need to identify other tautologies. How do you show that a formula is a tautology? When this is done "by hand" a "truth table" is constructed. A *truth table* is a disciplined approach to enumerating all the combinations of Boolean values for the variables in a formula

and for keeping track of the evaluations of subexpressions.

Truth tables are excruciatingly boring. No self-respecting computer science student who had to construct a lot of truth tables would do it by hand. That's why on Question 134, page 107 I challenged you write the function `tautp` to recognize tautologies for you. There are many tools on the web to check tautologies. Many companies sell *very expensive*, industrial-strength tools for checking tautologies.

Except on exams prohibiting the use of computers, you should use software to check unfamiliar tautologies!

But ok, because others will expect it of you, I'm going to show you a few truth tables and expect you to be able to do more.

To make a truth table, set up a column for each variable and each subformula, building up to the formula you want to check. That formula is in the far right-hand column. Then make a row for every combination of values of the variables. Then calculate and write into the cells the values of the respective subformulas in the corresponding cells. For brevity it is conventional to use T and F instead of $True$ and $False$.

Here is the truth table for one of the Basic tautologies, "$(p \wedge q) \rightarrow p$."

p	q	$(p \wedge q)$	$(p \wedge q) \rightarrow p$
T	T	T	T
T	F	F	T
F	T	F	T
F	F	F	T

Notice how the first two columns show all combinations of truth values for the variables p and q. Notice how the last column shows the truth value "$(p \wedge q) \rightarrow p$." And notice how the value in the last column is always T. The formula is $True$ under all combinations of values for its variables. It is a tautology.

Here is a non-tautology.

p	q	$(p \vee q)$	$(p \wedge q)$	$(p \vee q) \rightarrow (p \wedge q)$
T	T	T	T	T
T	F	T	F	F
F	T	T	F	F
F	F	F	F	T

Note that the last column contains some $False$s. This formula is not a tautology.

• **Question 166**: The formula checked above is the converse of one of our well-known tautologies. Which one? •

Here is the last one of these boring things I'm going to do! Below is a truth table to show that the transitivity of implication is really a tautology.

P	Q	R	$(P \rightarrow Q)$	$(Q \rightarrow R)$	$((P \rightarrow Q) \wedge (Q \rightarrow R))$	$(P \rightarrow R)$	$(((P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow (P \rightarrow R))$
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	T	T
F	T	F	T	F	F	T	T
F	F	T	T	T	T	T	T
F	F	F	T	T	T	T	T

• **QUESTION 167**: If you have a formula with n different variables in it, how many rows do you need in its truth table? •

• **QUESTION 168**: A femtosecond is 10^{-15} seconds. In order to perform one operation per femtosecond a sequential processor would have to operate at one million gigahertz – so a femtosecond is a *very fast* cycle length! Suppose you could fill in all the cells in one row of a given truth table in 1 femtosecond. Suppose the truth table was for a formula containing 1,000 propositional variables. About how long would it take to fill in the entire truth table? •

Today we have industrial-strength tools that can easily check formulas containing hundreds of thousands of variables. Sometimes they can check formulas containing many millions of variables. Use of such tools is standard practice in the hardware industry: microprocessors are essentially just big propositional formulas connected by memory devices. I'll explain that if you're interested.

How do computers check propositional formulas with millions of variables? They do *not* do it with truth tables!

4.6 About the Rules of Inference (Assignment 11: 7 days)

In this section, I discuss the informal proof from section 3.1 (page 83) again. I mimic the proof there, but I'll use the rules to carry out several of the "big steps" in that proof and I explain what I'm doing in great detail so that you understand exactly what these rules force you to do. If you don't understand the individual steps in each rule, you are liable to make logical mistakes. Please start by re-reading the succinct informal proof on page 87.

As you read this section, refer back pages 127–131 every time I mention a rule. You must learn to apply these rules flawlessly.

In the next section I present proofs in the notation you'll use most often but in this section I explore the minutia of each rule.

4.6.1 An Exploration of Rewrite

We want to prove

Theorem `first-app` (γ):

```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a)))
```

which I will refer to as formula γ and transform through γ', etc., to `t`.

We have the definitional axiom

Def Ax `app`:
```
(app x y)
=
(if (endp x)
    y
    (cons (first x)
          (app (rest x) y))).
```

Big Step 1: We'll use the Rewrite rule to replace `(app a b)` by the body of `app`. Re-read the Rewrite rule of inference, page 128. The rule requires that we carry out seven steps. In this application of the rule, ψ will be γ above and π is an address that points to the `(app a b)` in γ.

ψ:
```
(equal (first (app a b))
              ↑π
       (if (endp a)
           (first b)
           (first a)))
```

We factor goal ψ into the $True$ hypothesis (ψ_h) and the conclusion (ψ_c) shown above as ψ. Note that the term we want to rewrite is in the conclusion. Think about why we got the $True$ hypothesis. Hint: Re-read how to factor, page 113, and how to form the conjunction of a list of term, page 111.

The "previously proved theorem" ϕ we will use is the definitional axiom of `app`. We factor it into a hypothesis $True$ (ϕ_h) and the equality conclusion shown in Def Ax `app` (ϕ_c). We further factor ϕ_c into the pattern `(app x y)`, the replacement

β:
```
(if (endp x)
    y
    (cons (first x)
          (app (rest x) y)))
```

and the maintained equivalence `equal`.

Note that the term at π in ψ_c is `(app a b)` and that is a σ instance of the pattern, with σ being $\{\mathtt{x} \leftarrow \mathtt{a}; \mathtt{y} \leftarrow \mathtt{b}\}$.

Since the maintained equivalence is `equal`, we are not required to check that π admits a propositional replacement: we're doing a replacement of equals by equals and we can do that anywhere.

In step 6, we have to relieve the hypotheses of the theorem or axiom we're using, i.e., to

prove that the hypothesis of our goal implies the hypothesis of the axiom we're using. In this case, $\psi_h \rightarrow \phi_h/\sigma$ is just $True \rightarrow True$. Can we prove that? Yes! Think of it in its ACL2 form: `(implies t t)`. We can compute that it is equal to `t`. So the proof of this required theorem is just an application of the Computation rule of inference. By the Relieve-Hyps Waiver (page 129) we don't have to mention this.

So we obtain ψ'_c by replacing the term at π in ψ_c by β/σ, which produces:

ψ'_c:
```
(equal (first (if (endp a)
                  b
                  (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
```

Having carried out all seven steps required by the Rewrite rule, we create $True \rightarrow \psi'_c$ and refactor it any way we want to obtain our new goal ψ'. In our case, ψ' is just ψ'_c above. We call the final result γ':

γ':
```
(equal (first (if (endp a)
                  b
                  (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
```

That completes Big Step 1.

Big Step 2 is to move the `first` into the two branches of the first `if`. To carry out this step, we need a lemma – a little "stepping stone" theorem just to help us make the transformation. The lemma is

```
Lemma first-if:
(first (if u v w))
=
(if u (first v) (first w)).
```

We'll prove it later. For the moment, let's assume we have proved it.

• **QUESTION 169**: Describe how to use the Rewrite rule of inference to transform the above formula, γ', to

γ'':
```
(equal (if (endp a)
```

```
          (first b)
          (first (cons (first a)
                       (app (rest a) b))))
      (if (endp a)
          (first b)
          (first a)))
```

In particular, specify each of the 13 choices you have to make to apply the rule, by selecting, for each number from 1 to 13, one of the letters from A to Y corresponding to your choice.

Choices to make:

i	*choice*	(A–Y)
1.	ψ	____
2.	π	____
3.	ψ_h	____
4.	ψ_c	____
5.	ϕ	____
6.	ϕ_h	____
7.	ϕ_c	____
8.	pattern	____
9.	replacement	____
10.	maintained equivalence	____
11.	σ	____
12.	how step 6 is proved	____
13.	final refactoring of $\psi_h \rightarrow \psi'_c$	____

Possible selections

A. *True*
B. *False*
C. the formula of γ'
D. the formula of γ''
E. the formula of the `first-if` lemma
F. the formula defining `app`
G. the formula of the `first-cons` axiom
H. the occurrence of `(first (if ...))`
I. the first occurrence `(endp a)`
J. the occurrence of `(first (cons ...))`
K. `u`
L. `(if u v w)`
M. `(first v)`
N. `(first (if u v w))`
O. `(if u (first v) (first w))`
P. `equal` ("=")
Q. `iff` ("↔")
R. {`u` ← `(endp a)`, `v` ← `b`, `w` ← `a`}
S. {`u` ← `(endp a)`, `v` ← `(first b)`, `w` ← `(first a)`}
T. {`u` ← `(endp a)`, `v` ← `b`, `w` ← `(cons (first a) (app (rest a) b))`}
U. {`u` ← `(endp a)`, `v` ← `b`, `w` ← `(cons v (app a b))`}
V. De Morgan
W. Short-Circuit
X. Contraction
Y. Promotion

Some letters may be used multiple times; others won't be used at all. In the future I will not expect this level of detail when you use a rule! But I want to make sure that you have actually read and understood the rule. •

• **QUESTION 170**: Big Step 3 is to Rewrite again to produce γ''' below. Answer all three of these questions: What subterm of γ'' do we replace? What theorem or axiom do we use? What is the σ? •

The step above leaves us with:

γ''':
```
(equal (if (endp a)
           (first b)
           (first a))
       (if (endp a)
           (first b)
           (first a)))
```

Big Step 4 reduces this to *True*, via the Tautology Rule (page 128) using

Axiom `equal-reflexive`: `x = x`

Recall that "x = x" is just infix for `(equal x x)`.

That completes the proof of the `first-app` theorem – provided we prove the lemma `first-if` used above. I will show a succinct proof of `first-app` in the next section.

But while we're exploring the rules of inference, let's continue in this careful vein and look at the proof of the lemma used above.

4.6.2 An Exploration of Cases and Hypothesis

```
Lemma first-if:
(first (if u v w))
=
(if u (first v) (first w)).
```

To prove this, I'll appeal to the Cases rule. Re-read the Cases rule (page 129).

Our ψ is the `first-if` lemma above. I want to consider two cases: `u` is true or `u` is false.

Why did I decide on a case split and why this one? `First-if` tests `u` twice, once on the left-hand side and once on the right-hand side. By splitting on `u` I give myself the chance to get rid of those `if`-expressions. Watch.

To split on `u` as above, I need a disjunctive theorem like this:

$(x \leftrightarrow True) \vee (x \leftrightarrow False)$

that I can instantiate with `u` to obtain the two cases. (The rule allows an arbitrary number of cases, as long as their disjunction is a theorem.) Since we have the Basic tautology, $p \vee (\neg p)$, I'll use it.

In the Cases rule, let ϕ be $p \vee (\neg p)$. Let σ be $\{p \leftarrow \texttt{u}\}$. The disjuncts of ϕ/σ, called ϕ'_1 and ϕ'_2 in the statement of the rule, are `u` and `(¬ u)`. We have to prove $\phi'_1 \rightarrow \psi$ and $\phi'_2 \rightarrow \psi$.

Thus, we have two cases to deal with.

```
Case 1:
u → ((first (if u v w)) = (if u (first v) (first w))).
Case 2:
(¬ u) → ((first (if u v w)) = (if u (first v) (first w))).
```

To prove Case 1, we use the Hypothesis rule (page 129). Since this is the first time we've used it, we'll carry it out carefully.

We factor the formula of Case 1 into hypothesis `u`, and conclusion `((first (if u v w)) = (if u (first v) (first w)))`. The δ of the Hypothesis rule is `u`, the only hypothesis. We factor δ into pattern `u`, replacement `t`, and maintained equivalence `iff`. We let π be the address of the `u` in `(if u v w)`. Note that since the maintained equivalence is `iff`, we must confirm that π admits a propositional replacement. It does, because it points to the

test of an `if`. Therefore, we replace `u` by the replacement chosen in the factoring above, `t`. This produces

Case 1′:
`u` $\rightarrow$ `((first (if t v w)) = (if u (first v) (first w)))`.

We can then use Rewrite, with

Axiom `if-ax1`: `x` $\neq$ `nil` $\rightarrow$ `((if x y z) = y)`.

to reduce Case 1″ to:

Case 1‴:
`u` $\rightarrow$ `((first v) = (if u (first v) (first w)))`.

This application of Rewrite is actually a little interesting, because it is the first time we've applied Rewrite on a theorem with a hypothesis. Recall the "Relieving the Hypotheses" step, step 6 (page 128), of the Rewrite rule of inference. Step 6 says we have to prove the instantiated hypothesis of the theorem we're using, `if-ax1`, and we get whatever hypotheses govern the address we're rewriting. So our "proof obligation" under step 6 is `x` $\rightarrow$ `t` $\neq$ `nil`. We can reduce the conclusion of that implication to `t` by the Computation rule (page 130) and then Rewrite with the Short-Circuit tautology to reduce our proof obligation to $True$, thus "relieving the hypotheses." Again, the Relieve-Hyps Waiver allows us not to mention this.

To complete the proof of Case 1‴ we have to simplify the other side of the equality, `(if u (first v) (first w))`. That is done very similarly: use the Hypothesis rule to replace the `u` by `t` and then rewrite again with Axiom `if-ax1`. The proof obligation to relieve the hypotheses is the same as before: $True \rightarrow$ `t` $\neq$ `nil`.

• **Question 171**: Prove Case 2 at a similar level of detail. •

4.6.3 Another Exploration of Rewrite

Before moving on, let me show a non-trivial use of Rewrite. It is undoubtedly the most frequently used rule of inference in our collection.

Suppose we know this is a theorem:

`rev-rev`:
ϕ:
`(true-listp x)` $\rightarrow$ `((rev (rev x)) = x)`.

Suppose we're trying to prove:

ψ:
`((true-listp a)` $\wedge$ `(mem e (rev (rev (copy a)))))` $\rightarrow$ `(mem e a)`.

We want to use ϕ to simplify the `(rev (rev (copy a)))` to `(copy a)`. How do we do it?

First, we factor ψ so the hypothesis ψ_h is `((true-listp a)` $\wedge$ `(`$\neg$`(mem e a)))` and the conclusion, ψ_c is `(`$\neg$`(mem e (rev (rev (copy a)))))`. We do this because we must rewrite something in the conclusion. We'll see why in a moment.

The π we choose points to `(rev (rev (copy a)))` in ψ_c.

We factor theorem ϕ into hypothesis, ϕ_h, `(true-listp x)` and conclusion ϕ_c `(rev (rev x))=x`. We further factor ϕ_c into pattern `(rev (rev x))`, replacement `x`, and maintained equivalence `equal`. If we let σ be {`x` $\leftarrow$ `(copy a)`}, then we see that ϕ_c/σ is `(rev (rev (copy a)))`, which is indeed the term pointed to by π in ψ_c.

Since the maintained equivalence is `equal`, we don't have to check that π admits propositional replacement.

This brings us to step 6 of the use of Rewrite, the step in which we "relieve the hypotheses" of ϕ. We must prove

$\psi_h \rightarrow (\phi_h/\sigma)$:
`((true-listp a)` $\wedge$ `(`$\neg$`(mem e a)))` $\rightarrow$ `(true-listp (copy a))`.

This is not a trivial theorem; the Relieve-Hyps Waiver doesn't excuse us from explaining the proof. However, we're not in a position to prove this right now. What I would do in this situation is write a new lemma, perhaps the slightly simpler but sufficient:

`true-listp-copy:`
`(true-listp a)` $\rightarrow$ `(true-listp (copy a))`

and cite this lemma in my mention of Rewrite with `rev-rev`. I would then complete my proof of what I was working on and then do a proof of the lemma.

But I wanted to write down the proof obligation of this step 6 so you get a feel for what that step is all about. The theorem ϕ, does *not* tell us `(rev (rev x))` is equal to `x`! It tells us that `(rev (rev x))` is equal to `x` *provided* `(true-listp x)`. To use it to rewrite `(rev (rev (copy a)))` to `(copy a)`, we must prove that `(copy a)` is a `true-list`. When we prove that, we get to use as hypotheses all the hypotheses that are known to be true at address π. *That is why we factored ψ to put the target term in the conclusion!*

If we could prove the theorem required by step 6, then we could rewrite the term at π to `x`$/\sigma$, which is `(copy a)`. That would give us

ψ'_c:
`(`$\neg$`(mem e (copy a)))`

But we're not done yet! According to the Rewrite rule, we can return any refactoring of

$\psi_h \rightarrow \psi'_c$:
`((true-listp a)` $\wedge$ `(`$\neg$`(mem e a)))` $\rightarrow$ `(`$\neg$`(mem e (copy a)))`.

We could return that very formula. But since we started with

ψ:
`((true-listp a)` $\wedge$ `(mem e (rev (rev (copy a)))))` $\rightarrow$ `(mem e a)`

it would be nice to put things back where they were, by refactoring to

ψ':
`((true-listp a)` $\wedge$ `(mem e (copy a)))` $\rightarrow$ `(mem e a)`.

I would describe this proof step with:

`((true-listp a)` $\wedge$ `(mem e (rev (rev (copy a)))))` $\rightarrow$ `(mem e a)`
$\leftrightarrow$ {`rev-rev` and `true-listp-copy`}
`((true-listp a)` $\wedge$ `(mem e (copy a)))` $\rightarrow$ `(mem e a)`.

Note that since the Relieve-Hyp Waiver does not apply, I have to mention how I relieved the hypothesis.

Note that it looks like we used the Rewrite rule to rewrite one of our hypotheses. And we did! But we temporarily moved it into the conclusion so we could know what hypotheses we could use while relieving the hypotheses of ϕ in step 6. Then we put it back with the final refactoring.

Of course, we haven't finished the proof of ψ. To finish it, we'd have to prove ψ' above. But I don't do that here because I just wanted to illustrate the Rewrite rule of inference.

4.6.4 An Exploration of Constant Expansion

Let's prove that `(first (rest '(A B C)))` returns `B`. Of course, that's easy by the Computation rule. But I want to show you that we don't need the Computation rule to work with list constants. We can use the Constant Expansion ("Const") rule (page 130) and axioms.

Theorem `(first (rest '(A B C)))` = `'B`.
Proof:
lhs =
`(first (rest '(A B C)))`
= {Const)}
`(first (rest (cons 'A '(B C))))`
= {Const}
`(first (rest (cons 'A (cons 'B '(C)))))`
= {`rest-cons`}
`(first (cons 'B '(C)))`
= {`first-cons`}
`'B`
= *rhs*

□

• **Question 172**: "Compute" the value of `(app '(A) '(B C))` by doing a proof of `(equal (app '(A) '(B C)) '(A B C))`, without using the Computation rule. •

Computation is just the disciplined use of symbolic rewriting using the axioms and definitions.

I included the Computation rule of inference simply because it is convenient.

4.7 Writing Proofs Down (Assignment 11 cont'd)

Computer scientists give proofs in many different notations. If a professor asks you to give a proof, let's hope you've seen an example of his or her proof style! When writing a proof your goal is to make it clear to the reader that the formula is always true. Your goal is not to shoehorn your argument into some fixed format or to make the proof look like a secret language! You want the reader to understand your steps.

Here are several proofs of `first-app`. They're all identical in my opinion and I don't care which style you adopt.

Let's prove:

Theorem:

```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a))).
```

Here is the most straightforward translation of the proof we just did.

Proof 1:

```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a)))
=                                                          {app}
(equal (first (if (endp a)
                  b
                  (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
=                                                     first-if}
```

```
(equal (if (endp a)
           (first b)
           (first (cons (first a)
                        (app (rest a) b))))
       (if (endp a)
           (first b)
           (first a)))
=                                                        {first-cons}
(equal (if (endp a)
           (first b)
           (first a))
       (if (endp a)
           (first b)
           (first a)))
↔                              {Taut with Axiom equal-reflexive}
True.
□
```

The “box” is like saying “mate” in chess. It declares the game over. Some people write “Q.E.D.” instead, which stands for the Latin phrase “*quod erat demonstrandum*” meaning “that which was to be demonstrated.”

Classically, a proof starts with truths and manipulates them to derive the goal. Our proofs are backwards. We start with the goal we're trying to prove and manipulate it until we get *True*. Of course, the two systems are symmetric and most proofs actually take the form given here.

The proof above involves a lot of writing! With copy-and-paste editing it is easy to produce a proof like this: copy the last formula in the proof, paste it down, and then edit it slightly to perform your next rewrite and to add the justification. Then repeat.

But a closer inspection of our proof reveals that all of our rewrites, except the last, maintained `equal` and operated on the first argument of the goal equality. So here is another way to describe this same proof. Note that we write the theorem with an infix equality below, just to make the proof style more familiar.

Theorem:
`(first (app a b)) = (if (endp a) (first a) (first b)))`.

Proof 2: We will transform the left-hand side (*lhs*) into the right-hand side (*rhs*).

```
lhs
=
(first (app a b))
=                                                        {app}
(first (if (endp a)
         b
         (cons (first a)
```

```
               (app (rest a) b))))
=                                                            {first-if}
(if (endp a)
    (first b)
    (first (cons (first a)
                 (app (rest a) b))))
=                                                          {first-cons}
(if (endp a)
    (first b)
    (first a))
=
```

rhs

□

It should be clear to you that given a proof like Proof 2 we can produce a proof like Proof 1 using our rules.

In any case, these proofs are basically of the form "simplify the left-hand side to the right-hand side," e.g., $lhs = lhs_1 = lhs_2 = \ldots lhs_k = rhs$. Of course, sometimes it is easiest to simplify the right-hand side to the left-hand side. But most of the time you need to work on both sides; so I recommend just simplifying the two sides separately and then showing that the results are the same: "We first simplify the left-hand side: $lhs = lhs_1 = lhs_2 = \ldots = \alpha$. Now we simplify the right-hand side: $rhs = rhs_1 = rhs_2 = \ldots \alpha$. Notice they simplify to the same term. Hence $lhs = rhs$."

Sometimes your proofs are clearer if you skip over some steps! Some professors would actually prefer a "proof sketch" like this:

Theorem:

```
(equal (first (app a b))
       (if (endp a)
           (first b)
           (first a))).
```

Proof 3:
Use the definitions of `app` to rewrite the left-hand side to

```
(first (if (endp a)
           b
           (cons (first a)
                 (app (rest a) b))))
```

Then rewrite with `first-if` and Axiom `first-cons` to get

```
(if (endp a)
    (first b)
    (first a))
```

which is the right-hand side.

□

When you give a proof sketch like Proof 3, it is up to you to decide how big the steps can be. I've seen proof sketches like:

Proof 4:
Rewrite with the definitions of `app`, the lemma `first-if` and Axiom `first-cons` to get an instance of $x = x$. □

The trouble with such succinct sketches is that if you don't explicitly name a lemma you used then the reader can't follow it. But if you write some of the intermediate formulas, the reader might be able to figure out what you really did. This is important if your proofs are being graded!

To illustrate how to write a proof involving Cases and Hypothesis rules, let me prove this useful theorem about `if`:

Theorem `if-swap`:
`(if (not a) b c) = (if a c b).`

Proof 5:

Case 1:
`a` → `((if (not a) b c) = (if a c b)).`
↔ {Hyp (twice)}
`a` → `((if (not t) b c) = (if t c b)).`
↔ {Comp}
`a` → `((if nil b c) = (if t c b)).`
↔ {`if-ax2`}
`a` → `(c = (if t c b)).`
↔ {`if-ax1`}
`a` → `(c = c).`
↔ {`equal-reflexive`}
`a` → *True*.
↔ {Short-Circuit}
True

Case 2:
(¬ `a`) → `((if (not a) b c) = (if a c b)).`
↔ {Hyp (twice)}
(¬ `a`) → `((if (not nil) b c) = (if nil c b)).`
↔ {Comp}
(¬ `a`) → `((if t b c) = (if nil c b)).`
↔ {`if-ax1`}
(¬ `a`) → `(b = (if nil c b)).`
↔ {`if-ax2` }
(¬ `a`) → `(b = b).`

$\leftrightarrow$ {`equal-reflexive`}
`(¬ a)` $\rightarrow$ *True*.
$\leftrightarrow$ {Short-Circuit}
True
$\square$

This proof, as detailed as it is, omits certain simple steps, like relieving the hypothesis of Axiom `if-ax1`, which requires that we prove *True* $\rightarrow$ `t` $\neq$ `nil`. I'm content to ignore the omission of such simple steps – computation on constants and uses of such things as the Basic and Short-Circuit tautologies.

Eventually in this course, a proof like Proof 5 will be written *really succinctly*: "Case split on `a` and simplify." But not yet!

Often when proving an implication, whether one generated by the Cases rule or otherwise, it is often useful to pull the hypotheses out, give them names, and not keep writing them down. For example, to prove $(\alpha_1 \wedge \alpha_2) \rightarrow \beta$ I might write this:

Proof:
H1: α_1
H2: α_2

I will reduce β to *True* under these hyps.
β
$\leftrightarrow$ $\{\ldots\}$
β_1
...
$\leftrightarrow$ $\{\ldots\}$
True
$\square$

and in the justifications refer to H1 and H2 to identify the hypotheses I'm using.

Using this notation I can make the proof of `if-swap` more succint. Note below that instead of naming the hypotheses (`a` in one case and `(¬ a)` in the other) I label them with "Case 1" or "Case 2". I don't actually use their names below because there is only one hypothesis in each case, so it is clear which hypothesis the Hyp rule refers to.

Theorem `if-swap`:
`(if (not a) b c) = (if a c b).`

Proof 6:

Case 1: `a`
`(if (not a) b c) = (if a c b).`
$\leftrightarrow$ {Hyp (twice)}
`(if (not t) b c) = (if t c b).`
$\leftrightarrow$ {Comp}
`(if nil b c) = (if t c b).`

$\leftrightarrow$ {`if-ax2`}
`c = (if t c b).`
$\leftrightarrow$ {`if-ax1`}
`c = c.`
$\leftrightarrow$ {`equal-reflexive`}
True.

Case 2: `(¬ a)`
`(if (not a) b c) = (if a c b).`
$\leftrightarrow$ {Hyp (twice)}
`(if (not nil) b c) = (if nil c b).`
$\leftrightarrow$ {Comp}
`(if t b c) = (if nil c b).`
$\leftrightarrow$ {`if-ax1`}
`b = (if nil c b).`
$\leftrightarrow$ {`if-ax2`}
`b = b.`
$\leftrightarrow$ { `equal-reflexive`}
True.
□

Note that we didn't keep writing "`a` → ..." or "`(¬ a)` → ..." the way we did in Proof 5. This can save a lot of writing if you split on a big formula instead of a variable like `a`.

Let's prove

Theorem `true-listp-nil`:
`((true-listp x) ∧ (endp x)) → (x=nil).`

Proof:
Given
H1: `(true-listp x)`
H2: `(endp x)`

I will prove
Concl: `x=nil`

H1
= {`true-listp`}
```
(if (endp x)
    (equal x nil)
    (true-listp (rest x)))
```
= {Hyp H2}
```
(if t
    (equal x nil)
    (true-listp (rest x)))
```
= {`if-ax1`}
`(equal x nil)`

=
H1'

Now, I turn my attention to Concl:
`Concl`
=
`x=nil`
$\leftrightarrow$ {Hyp H1'}
True
□

Notice how I rewrote hypothesis H1 (with the definition of `true-listp` and using hypothesis H2) to get a "new" hypothesis which I named H1'. Then I used H1' in my proof that Concl $\leftrightarrow$ *True*. Remember that if you're rewriting a hypothesis you can use the other hypotheses as hypotheses, and you can use the *negation* of the conclusion as a hypothesis (but I didn't above).

• **Question 173**: Explain why you can use the negation of the conclusion of your goal as a hypothesis when using the Hyp rule with an address that points into a hypothesis of your goal. •

Warning: There is a danger many students fall into when writing proofs this way. Below I "prove" the non-theorem `((x=y) → (1=2))`. You can see this is not always true: let `x` be `1` and let `y` be `1`. In this environment `((x=y) → (1=2))` $\Longrightarrow$ `nil`. So I shouldn't be able to prove it!

Non-Theorem!
`(x=y) → (1=2)`
Proof:
H1: `x=y`

Concl: `1=2`
$\leftrightarrow$ {Hyp H1 with $\sigma = \{\mathtt{x} \leftarrow \mathtt{1}, \mathtt{y} \leftarrow \mathtt{2}\}$}
True
□?

The mistake in this "proof" is that the Hyp rule has no σ: you can't instantiate the (free) variables[5]. I think this mistake happens because students confuse the Hypothesis rule with the Rewrite rule. In the Rewrite rule, you're using a previously proved theorem, known to be true for all values of the variables, so you can instantiate the (free) variables in that theorem.

When the conclusion we're manipulating is an equality or propositional equivalence, we might use the "$lhs = \ldots = rhs$" or "$lhs \leftrightarrow \ldots \leftrightarrow rhs$" style within this notation. Here's a proof outline that mixes the idea of "Givens", cases, and a chain of equalities.

[5]Eventually we will introduce notation that involves variables you may instantiate. I'll explain in Chapter 7.

Theorem: $(\alpha_1 \wedge (\alpha_2 \wedge \alpha_3)) \rightarrow (\delta = \gamma)$

Proof:
I'll take the hypotheses as given and transform the left-hand side of the conclusion to the right-hand side.
Given
H1: α_1
H2: α_2
H3: α_3

`Let`
lhs: δ
rhs: γ

To reduce the *lhs* to the *rhs*, I will case split on β:

Case 1 β
lhs $= \delta$
$=$ { *justification* }
...
$= \gamma$
$= rhs$

Case 2 $(\neg\beta)$
lhs $= \delta$
$=$ { *justification* }
...
$= \gamma$
$= rhs$
□

You will undoubtedly have to make up additional proof notation to explain some of your proofs. Just remember that the point of a proof is to establish that something is true – and usually to convince another person that your argument is correct. It is helpful if you say what you're doing or planning to do in a proof. *Don't feel compelled to make your proof look like some secret language!*

4.8 Proofs of Tautologies (Assignment 11 cont'd)

We've seen that tautologies can be established using truth tables. In fact, a truth table is just a succinct way to present a proof by Cases, in which you case split on every variable. Suppose you're trying to prove some propositional formula involving just the variables `p` and `q`, e.g., $(\phi$ `p q`$)$. An outline of a proof by Cases would be:

Proof:

```
Case 1 p:
  Case 1.1 q:
  (φ t t)= t                                    {Hyps and Comp}
  Case 1.2 (¬ q):
  (φ t nil) = t                                 {Hyps and Comp}
Case 2 (¬ p):
  Case 2.1 q:
  (φ nil t)  = t                                {Hyps and Comp}
  Case 2.2 (¬ q):
  (φ nil nil) = t                               {Hyps and Comp}
□
```

Basically we split on the variables and then used the Hyp rule repeatedly to replace the variables by `t` or `nil` according to whichever case we're in, and then compute the value with the Comp rule. That's just the truth table method in proof form.

But you don't have to use truth tables or exhaustive case analysis to prove propositional tautologies! You can prove them by symbolic manipulation.

• **QUESTION 174**: Prove the following theorem:

```
((A ∧ ((B ∨ C) ∧ (D → (¬(E ∧ (G ∧ H))))))
 ∧
 ((U ∨ V) ↔ (X ∧ (¬(X ∨ Y)))))
→
(A ∧ ((B ∨ C) ∧ (D → (¬(E ∧ (G ∧ H))))))
```

Hint: A truth table will have 2^{12} rows. Maybe you should consider using a tautology. •

Sometimes the easiest way to prove a tautology is to rearrange it using other tautologies to show that it is an instance of some tautology. But to do that you have to be good at using tautologies to rearrange formulas.

• **Question 175**: Prove the following by using tautologies with the Rewrite rule of inference.

`(D ∧ ((A ∧ C) ∧ B)) ↔ (A ∧ (B ∧ (C ∧ D))).` •

• **Question 176**: Prove the following by using tautologies with the Rewrite rule of inference.

`((D ∨ C) ∨ (B ∨ A)) ↔ (A ∨ (B ∨ (C ∨ D))).` •

I assume that given any nest of `and`s on a given set of variables you can rearrange it to any equivalent set of `and`s on those variables. Same for `or`s.

And so from now on we're not necessarily going to write "∧" and "∨" with just two arguments. That is, I might write

`(D ∧ (¬ A) ∧ C ∧ (¬ B))`

and then say I've applied De Morgan to get

`(D ∧ C ∧ (¬(A ∨ B))).`

What did I really do? I did this:

```
(D ∧ (¬ A) ∧ C ∧ (¬ B))
↔                                   {Associativity and Commutativity}
(D ∧ C ∧ ((¬ A) ∧ (¬ B)))
↔                                   {De Morgan}
(D ∧ C ∧ (¬(A ∨ B))).
```

That is, in the future I'm going to use Associativity and Commutativity of "$\wedge$" and "$\vee$" freely and implicitly. But I assume you can fill in the gaps! I'll also use Double Negation freely and implicitly.

Let's prove the following tautology, without using a truth table.

Theorem.
`(¬C ∧ ((B → (¬ C)) → A)) → ((A → C) → ` *False*`)`

I don't like thinking about implications involving implications, i.e., things of the form $(p \to (q \to r))$ or of the form $((p \to q) \to r)$. The theorem above has subformulas of each of these two forms in it!

I know that I can transform subformulas of the first form $(p \to (q \to r))$ to $(p \wedge q) \to r$. This is called Promotion and is justified by a tautology of that name on page 131. This tranformation is good because it gets rid of an implication and gives me another hypothesis, q, to work with as I work on r.

As for the other form, $((p \to q) \to r)$, there's not a tautology in our list to handle it. But I can prove one. I'll prove this as a lemma and then return to our theorem.

Lemma `double-implies:`
`((p → q) → r) ↔ (((¬ r) → p) ∧ ((¬ r) → (¬ q)))`
Proof:
I will transform the left-hand side into the right-hand side.
$lhs =$

```
 ((p → q) → r)
↔                                   {Implicative Disjunction}
 (((¬ p) ∨ q) → r)
↔                                   {Implicative Disjunction}
 ((¬((¬ p) ∨ q)) ∨ r)
↔                                   {De Morgan}
 ((p ∧ (¬ q)) ∨ r)
↔                                   {Distributivity}
 ((p ∨ r) ∧ ((¬ q) ∨ r))
↔                                   {Implicative Disjunction (twice)}
 (((¬ r) → p) ∧ ((¬ r) → (¬ q)))
```

$= rhs$

□

• **Question 177**: Show a step in the proof above in which Commutativity was used implicitly. •

• **Question 178**: Show a step in the proof above in which Double Negation was used implicitly. •

Now let's return to the theorem we were working on.

Theorem.
$((\neg C) \wedge ((B \rightarrow (\neg C)) \rightarrow A)) \rightarrow ((A \rightarrow C) \rightarrow False)$

Proof:
$((\neg C) \wedge ((B \rightarrow (\neg C)) \rightarrow A)) \rightarrow ((A \rightarrow C) \rightarrow False)$
$\leftrightarrow$ {double-implies}
$((\neg C) \wedge ((\neg A) \rightarrow B) \wedge ((\neg A) \rightarrow C)) \rightarrow ((A \rightarrow C) \rightarrow False)$
$\leftrightarrow$ {Promotion}
$((\neg C) \wedge ((\neg A) \rightarrow B) \wedge ((\neg A) \rightarrow C) \wedge (A \rightarrow C)) \rightarrow False$
$\leftrightarrow$ {Contraposition}
$((\neg C) \wedge ((\neg A) \rightarrow B) \wedge ((\neg C) \rightarrow A) \wedge (A \rightarrow C)) \rightarrow False$
$\leftrightarrow$ {Forward Chaining}
$((\neg C) \wedge ((\neg A) \rightarrow B) \wedge A \wedge (A \rightarrow C)) \rightarrow False$
$\leftrightarrow$ {Forward Chaining}
$((\neg C) \wedge ((\neg A) \rightarrow B) \wedge A \wedge C) \rightarrow False$
$\leftrightarrow$ {Contraposition }
$(((\neg A) \rightarrow B) \wedge A \wedge C) \rightarrow C$
$\leftrightarrow$ {Taut Basic}
True
□

• **Question 179**: Explain what really happened on the first Forward Chaining step in the proof above. •

You will be expected to be able to explain any step in a proof.

Now we will prove our theorem again, only this time we will use case analysis too.

Theorem.
$((\neg C) \wedge ((B \rightarrow (\neg C)) \rightarrow A)) \rightarrow ((A \rightarrow C) \rightarrow False)$

Proof:

Case 1 C:
$((\neg C) \wedge ((B \rightarrow (\neg C)) \rightarrow A)) \rightarrow ((A \rightarrow C) \rightarrow False)$
$\leftrightarrow$ {Hyp Case 1}
$((\neg True) \wedge ((B \rightarrow (\neg True)) \rightarrow A)) \rightarrow ((A \rightarrow True) \rightarrow False)$
$\leftrightarrow$ {Comp}
$(False \wedge ((B \rightarrow (\neg True)) \rightarrow A)) \rightarrow ((A \rightarrow True) \rightarrow False)$

$\leftrightarrow$ {Short-Circuits}
$True$

Case 2 ($\neg$ C):
$((\neg \mathtt{C}) \wedge ((\mathtt{B} \rightarrow (\neg \mathtt{C})) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow \mathtt{C}) \rightarrow False)$
$\leftrightarrow$ {Hyp Case 2}
$((\neg False) \wedge ((\mathtt{B} \rightarrow (\neg False)) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow False) \rightarrow False)$
$\leftrightarrow$ {Comp}
$(True \wedge ((\mathtt{B} \rightarrow True) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow False) \rightarrow False)$
$\leftrightarrow$ {Short-Circuits}
$(True \wedge (True \rightarrow \mathtt{A})) \rightarrow (\neg \mathtt{A}) \rightarrow False$
$\leftrightarrow$ {Short-Circuits}
$\mathtt{A} \rightarrow \mathtt{A}$
$\leftrightarrow$ {Basic}
$True$
$\square$

Note that when I used the Hyp rule I cited "Case 1" or "Case 2" meaning I used the hypothesis introduced by that case.

The repeated uses of Short-Circuit and Computation are so monotonous that I'll frequently abbreviate them as "Prop Simp" meaning *propositional simplification.* I show intermediate steps only so you can follow along. I recommend that you show intermediate steps when you do it too. Here's that same proof, somewhat more succinctly.

Theorem.
$((\neg \mathtt{C}) \wedge ((\mathtt{B} \rightarrow (\neg \mathtt{C})) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow \mathtt{C}) \rightarrow False)$

Proof:
Case 1 C:
$((\neg \mathtt{C}) \wedge ((\mathtt{B} \rightarrow (\neg \mathtt{C})) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow \mathtt{C}) \rightarrow False)$
$\leftrightarrow$ {Hyp Case 1}
$((\neg True) \wedge ((\mathtt{B} \rightarrow (\neg True)) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow True) \rightarrow False)$
$\leftrightarrow$ {Prop Simp}
$True$
Case 2 ($\neg$ C):
$((\neg \mathtt{C}) \wedge ((\mathtt{B} \rightarrow (\neg \mathtt{C})) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow \mathtt{C}) \rightarrow False)$
$\leftrightarrow$ {Hyp Case 2}
$((\neg False) \wedge ((\mathtt{B} \rightarrow (\neg False)) \rightarrow \mathtt{A})) \rightarrow ((\mathtt{A} \rightarrow False) \rightarrow False)$
$\leftrightarrow$ {Prop Simp}
$(True \wedge (True \rightarrow \mathtt{A})) \rightarrow ((\neg \mathtt{A}) \rightarrow False)$
$\leftrightarrow$ {Prop Simp}
$\mathtt{A} \rightarrow \mathtt{A}$
$\leftrightarrow$ {Basic}
$True$
$\square$

By doing a case analysis on the "right" variable you can sometimes drastically simplify the two formulas you have to prove. Doing it on the "wrong" variable can help very little.

• **Question 180**: Prove the theorem just proved again, by doing a case analysis on `A` instead of `C`. •

Commercial software systems designed to recognize tautologies containing thousands or millions of variables use a combination of case splitting and rewriting.

• **QUESTION 181**: Fill in the eight blanks below to make the following a proof. You may use the propositional identities on page 131. Just turn in a list of numbers from 1–8 with the appropriate content for each blank.

Theorem

```
(((A → (B ∨ C))
  ∧
  A
  ∧
  ((¬ D)→(¬ B))
  ∧
  (C → D))
 →
 D)
```

Proof:

```
(((A → (B ∨ C))
  ∧
  A
  ∧
  ((¬ D)→(¬ B))
  ∧
  (C → D))
 →
 D)
```

↔ {____1____}

```
(((B ∨ C)
  ∧
  A
  ∧
  ((¬ D)→(¬ B))
  ∧
  (C → D))
 →
 D)
```

↔ {Hyp (twice)}

```
(((B ∨ C)
  ∧
  A
  ∧
  ((¬ False)→(¬ B))
  ∧
  (C → ____2____))
 →
 D)
```

↔ {Computation, ____3____ (twice)}

```
(((B ∨ C)
  ∧
  A
  ∧
  (¬ B)
  ∧
  (¬ C))
 →
 D)
```

↔ {Hyp}

```
(((____4____ ∨ C)
  ∧
  A
  ∧
  (¬ B)
  ∧
  (¬ C))
 →
 D)
```

↔ {Short-Circuit}

```
((____5____
  ∧
  A
  ∧
  (¬ B)
  ∧
  (¬ C))
 →
 D)
```

↔ {Hyp}

((___6___
  ∧
  A
  ∧
  (¬ B)
  ∧
  (¬ C))
 →
 D)

↔ {___7___}

(*False*
 →
 D)

↔ {___8___}

True
□ •

4.9 Counterexamples, Witnesses, and Models (Assignment 12: 14 days)

Proofs are the best way to show that a formula ϕ is valid or always true.

Sometimes you might want to show that a formula is not valid. That is, you might want to show that there are values of the variables that make the formula evaluate to false. That's called a "counterexample" to the formula. Technically, a *counterexample* is a substitution σ such that ϕ/σ contains no variables and evaluates to false. (Sometimes counterexamples are not exhibited as substitutions but just as equalities that specify the values of the variables, e.g., "let `x` be `7` and `y` be `-7`.")

Sometimes formula ϕ contains undefined functions. For example, you might someday wish to add a new axiom corresponding to the informal remark "suppose `f` returns a natural greater than its argument." E.g., `(and (natp (f n)) (< n (f n)))`. You might want to establish that there really *is* such a function! You do that by providing a *witness* for the undefined function in ϕ that makes it a theorem. More generally the new axiom might contain several undefined functions and a collection of witnesses for them that make the formula a theorem is called a *satisfying model* or simply *model* of the formula.

For example, to model the proposed axiom about `f` above I might use `(defun f (x) (if (zp n) 1 (+ 1 n)))`. It is useful to model your axioms to make sure you're not talking nonsense. For example, if I said "suppose `f` returns a natural number between its two arguments" and I formalized it with

```
(and (natp (f i j))
     (< i (f i j))
     (< (f i j) j))
```

then the logic makes no sense. Why? There is no such function `f`. How do I know? If I had the formula above as an axiom then I could instantiate it to get:

```
(and (natp (f 3 3))
     (< 3 (f 3 3))
     (< (f 3 3) 3))
```

and derive `(< 3 3)`. But that formula evaluates to $False$, so I've just proved $False$. From $False$ I can derive anything.

Sometimes you might wish to show that a formula ϕ containing undefined functions is not valid (i.e., is *invalid*). You usually can't exhibit a counterexample because you cannot compute the values of the undefined functions. To show that such a ϕ is invalid you have to provide example definitions for the undefined functions and then show a counterexample that makes ϕ evaluate to false under those definitions. Such a collection of definitions is called a *countermodel*.

4.10 Practice Proofs (Assignment 12 cont'd)

Throughout this whole book, when I state a question as a formula, I mean for you to prove it or exhibit a counterexample and/or a countermodel.

Many of the formulas below are *very useful* in your later proofs. I recommend that you read and think about every question, even those not assigned for homework. You should learn to rely on these theorems.

When I follow a question's number with a symbol, that symbol is the name I'll use for that formula later. I find the names easier to remember than the numbers.

In this section, please write your proofs at the level of detail shown in Proofs 1, 2, 5, and 6 of the previous section. In your proofs you may use any of the previously mentioned theorems by name (or number). You may wish to state and prove lemmas to help simplify your proofs. But do not use truth tables below. If you want to prove a tautology, do it symbolically, by using other tautologies to rearrange formulas. The unfamiliar function symbols, `f`, `g`, etc., may be assumed to be legitimate functions of the appropriate arity; you just don't know their definitions. For your convenience, at the end of the section I've included the definitions of the defined functions mentioned below.

• **Question 182**: `(if-t)`

`(if t x y)` = `x` •

• **Question 183**: (`if-nil`)

`(if nil x y) = y` •

• **Question 184**: (`if-if`)

`(if (if a b c) x y) = (if a (if b x y) (if c x y))` •

• **Question 185**: (`if-distrib`)

`(f x (if a b c)) = (if a (f x b) (f x c))`

You do not need to know a definition for `f` to prove this theorem. Note that in general you can distribute `if` over any function symbols as suggested by the theorem above. •

• **Question 186**: (`if-split`)

`(if p q r)` $\leftrightarrow$ `((p` $\rightarrow$ `q)` $\wedge$ `((`$\neg$ `p)` $\rightarrow$ `r))` •

Note that questions 184–186 establish that if we expand terms in a formula and get some `if`s we can distribute the function symbols over the `if`s to lift all the `if`s to the top and then we can split the goal into a conjunction of new formulas.

• **Question 187**: (`endp-nil`)

`(endp nil) = t` •

• **Question 188**: (`endp-cons`)

`(endp (cons x y)) = nil` •

• **Question 189**: (`consp-endp-nil`)

`(consp x)` $\rightarrow$ `((endp x) = nil)` •

• **Question 190**: (`consp-endp-t`)

`(`$\neg$`(consp x))` $\rightarrow$ `((endp x) = t)` •

• **Question 191**: (`not-endp-consp`)

`(`$\neg$`(endp x))` $\rightarrow$ `((consp x) = t)` •

• **Question 192**: (`first-7`)

`(first 7)= nil` •

• **Question 193**: (`rest-nil`)

`(rest nil) = nil` •

• **QUESTION 194**: (`app-cons`)

`(app (cons e a) b) = (cons e (app a b))`

Make sure you type only well-formed terms in your proof, and format them sensibly! •

• **Question 195**: (`app-of-non-endp`)

`(¬(endp x)) → ((app x y) = (cons (first x) (app (rest x) y)))`

The relevant definitions are below. •

• **Question 196**: (`app-commutative-wrong!`) The following "identity" is invalid. Provide a counterexample.

`(app a b) = (app b a)`

The relevant definitions are below. •

• **Question 197**: (`app-left-id`)

`(app nil b) = b`

The relevant definitions are below. •

• **Question 198**: (`app-right-id-wrong1`) The following "identity" is invalid. Provide a counterexample.

`(app a nil)= a`

The relevant definitions are below. •

• **QUESTION 199**: (`app-right-id-wrong2`) The following "identity" is invalid. Provide a counterexample.

`(¬(endp a)) → ((app a nil)= a)`

The relevant definitions are below. You may just type the counterexample (you do not have to use ACL2 to evaluate the formula). •

• **Question 200**: (`true-listp-cons`)

`(true-listp y) → (true-listp (cons x y))`

The relevant definitions are below. •

• **Question 201**: (`mem-singleton`)

```
(mem e (cons d nil)) ↔ ((e = d))
```

The relevant definitions are below. •

• **Question 202**:

```
(e ≠ d) ∧ (mem e (cons d x)) → (mem e x)
```

The relevant definitions are below. •

• **Question 203**:

```
    (((consp x) → (stringp y))
  ∧
    ((symbolp z) ∧ (consp x)))
→
  (stringp y) •
```

• **Question 204**:

```
  ((P → Q)
   ∧
   (P → R))
→
  (P → (Q ∧ R)) •
```

• **Question 205**:

```
  ((A → (B ∨ C))
  ∧
   (A ∨ C)
  ∧
   (C → (¬ B)))
→
  C  •
```

• **Question 206**:

```
  ((A → D)
  ∧
   (B → E)
  ∧
   (D → (E → C)))
→
```

```
  ((A ∧ B) → C) •
```

• **Question 207**:

```
((R ∧ Q) → (Q → R))  •
```

• **Question 208**:

```
((A → B) ∨ (B → A)) •
```

• **Question 209**:

```
((R → (¬ P)) → ((P ∧ R) → S))  •
```

• **Question 210**:

```
((R ∧ (((¬ P) → (¬ R)) ∧ (P → (S ∨ U)))) → (S ∨ U)) •
```

• **Question 211**:

```
(((A → (B ∨ C)) ∧ (A ∧ (((¬ D) → (¬ B)) ∧ (C → D)))) → D) •
```

• **Question 212**:

```
  ((A → (B ∨ C))
  ∧
   (D → A))
→
  (B ∨ (D → C)) •
```

• **Question 213**:

```
   (endp a)
→
   ((app (app a b) c) = (app a (app b c))) •
```

• **Question 214**:

```
    (¬(endp a)
  ∧
     ((app (app (rest a) b) c) = (app (rest a) (app b c))))
→
  ((app (app a b) c) = (app a (app b c))) •
```

The relevant definitions are below. •

• **QUESTION 215**: Fill in the seven blanks below to make this a proof. Just turn in a list

of numbers from 1–7 with the appropriate content for each blank. The relevant definitions are below. As usual, take care with your syntax.

Theorem:

```
   (endp a)
 →
   ((true-listp a) → ((app a nil) = a))
```

Proof:

```
   (endp a)
 →
   ((true-listp a) → ((app a nil) = a))
```

↔ {Promotion}

```
  (____1____)
 →
  ((app a nil) = a)
```

↔ {def `true-listp`}

```
  ((endp a)
  ∧
   ____2____)
 →
  ((app a nil) = a)
```

↔ {____3____ and ____4____}

```
  ((endp a)
  ∧
   (equal a nil))
 →
  ((app a nil) = a)
```

↔ {Hyp (twice)}

```
  ((endp a)
  ∧
   (equal a nil))
 →
  ____5____
```

↔ {Computation}

```
  ((endp a)
  ∧
   (equal a nil))
 →
  ____6____
```

$\leftrightarrow$ {___7___}

True

□ •

• **QUESTION 216**: Fill in the nine blanks below to make this a proof. Just turn in a list of numbers from 1–9 with the appropriate content for each blank. As usual, take care with your syntax. The relevant definitions are below.

Theorem:

```
   ((¬ (endp a))
    ∧
    ((true-listp (rest a)) → ((app (rest a) nil) = (rest a))))
 →
   ((true-listp a) → ((app a nil) = a))
```

Proof:

```
   ((¬ (endp a))
    ∧
    ((true-listp (rest a)) → ((app (rest a) nil) = (rest a))))
  →
   ((true-listp a) → ((app a nil) = a))
```

$\leftrightarrow$ {___1___}

```
   ((¬ (endp a))
    ∧
    ((true-listp (rest a)) → ((app (rest a) nil) = (rest a)))
    ∧
    ___2___)
  →
   ((app a nil) = a)
```

$\leftrightarrow$ {___3___}

```
   ((¬ (endp a))
    ∧
    ((true-listp (rest a)) → ((app (rest a) nil) = (rest a)))
    ∧
    (if (endp a)
        (equal a nil)
        (true-listp (rest a))))
  →
   ((app a nil) = a)
```

$\leftrightarrow$ {___4___ and ___5___}

```
   ((¬ (endp a))
```

```
    ∧
    ((true-listp (rest a)) → ((app (rest a) nil) = (rest a)))
    ∧
    (true-listp (rest a)))
  →
   ((app a nil) = a)

↔                                              {Forward Chaining }

   ((¬ (endp a))
    ∧
    ____6____
    ∧
    (true-listp (rest a)))
  →
   ((app a nil) = a)

↔                                              {app-of-non-endp}

   ((¬ (endp a))
    ∧
    ((app (rest a) nil) = (rest a))
    ∧
    (true-listp (rest a)))
  →
   (____7____ = a)

↔                                              {____8____}

   ((¬ (endp a))
    ∧
    ((app (rest a) nil) = (rest a))
    ∧
    (true-listp (rest a)))
  →
   ((cons (first a) (rest a)) = a)

↔                          {Axiom cons-first-rest (page 125)}

   ((¬ (endp a))
    ∧
    ((app (rest a) nil) = (rest a))
    ∧
    (true-listp (rest a)))
  →
   (a = a)

↔                          {Axiom ____9____ and Short-Circuit }
```

True
□ •

• **Question 217**:

```
    ((a = b)
  ∧
    ((p a) → ((f a) = (g a))))
→
  ((p b) → ((h (f b)) = (h (g b))))
```

You do not need definitions for `f`, `g`, `h`, and `p`. •

• **Question 218**:

```
   (endp a)
→
   ((true-listp a) → ((rev (rev a)) = a))
```

The relevant definitions are below. *Please do not use* `rev-rev`, *i.e., the theorem* `(true-listp x) → ((rev (rev x)) = x)`, *in your proof!* •

• **QUESTION 219**: Give a proof of the following, comparable to the proof of sketched in Question 216 (page 164). Take care with your syntax; remember, your job is to provide a clear and convincing proof.

```
    ((¬(endp a))
  ∧
     ((true-listp (rest a)) → ((rev (rev (rest a))) = (rest a))))
→
   ((true-listp a) → ((rev (rev a)) = a))
```

The relevant definitions are below. In addition, you may use the following theorems in your proof:

```
not-endp-true-listp:
(¬ (endp x)) → ((true-listp x) = (true-listp (rest x)))

not-endp-rev:
(¬ (endp x)) → ((rev x) = (app (rev (rest x)) (cons (first x) nil)))

app-singleton:
(app (cons x nil) y) = (cons x y)

rev-singleton:
(rev (cons x nil)) = (cons x nil)

rev-app:
(rev (app x y)) = (app (rev y) (rev x))
```

Please do not use `rev-rev`*, i.e., the theorem* `(true-listp x)` $\rightarrow$ `((rev (rev x))` $=$ `x)`*, in your proof!* •

The definitions below are listed alphabetically.

```
(defun app (x y)
  (if (endp x)
      y
      (cons (first x)
            (app (rest x) y))))


(defun mem (x y)
  (if (endp y)
      nil
      (if (equal x (first y))
          t
          (mem x (rest y)))))


(defun rev (x)
  (if (endp x)
      nil
      (app (rev (rest x))
           (cons (first x) nil))))


(defun true-listp (x)
  (if (endp x)
      (equal x nil)
      (true-listp (rest x))))
```

5

Structural Induction (Assignment 13: 12 days)

We have seen how to prove some very simple theorems about recursive functions. Suppose we have:

```
(defun f (x)
  (if (endp x)
      t
      (f (rest x))))
```

Then you should be able to prove:

• **Question 220**:

`(f (cons e a))` = `(f a)` •

And I bet your intuition tells you the answer to this question:

• **Question 221**: What value does `(f x)` return? •

I suspect you're thinking "`(f x)` can't return anything but `t`!" Or, to be a little more operational, "`(f x)` scans down `x` til it gets to the end and then returns `t`."

But the logical machinery we have set up doesn't allow you to prove what your intuition rightly tells you. We can do case analysis *ad infinitum*, on whether `(endp x)`, `(endp (rest x))`, `(endp (rest (rest x)))`, etc. And in each case we can use the definition of `f` to rewrite `(f x)` to `t`. But we canot produce a finite number of cases to cover all possible `x`.

To cope with this – to prove interesting things about recursive functions – we must introduce a new rule of inference that mimics your intuition. Expressed in terms of `(f x)` above, the intuition goes like this: "I think `f` always returns `t`. It certainly does when it takes the easy way out. And if the recursive call, `(f (rest x))`, returns `t`, then the main branch returns `t` too. So I'm right: `f` can't do anything but return `t`."

In this section we state a rule of inference, called "induction," that formalizes this kind of thinking. It is absolutely fundamental to computer science because virtually everything in CS involves repeatedly doing something and we must understand what the ultimate result will be. Induction is *the* mathematical tool for thinking about repetition and recursion.

The version of induction we state below is a *very* special case of a much more general idea. But we don't deal with generalized induction in this course.

5.1 New Rule of Inference (Assignment 13 cont'd)

We say δ is a `rest`-*nest around* v if δ is `(rest` v`)`, or δ is a `rest`-nest around `(rest` v`)`.

Thus, all of the following are `rest` nests around `x`: `(rest x)`, `(rest (rest x))`, `(rest (rest (rest x)))`.

A substitution σ is a `rest`-nest *substitution on* v if the image of v under σ is a `rest`-nest around v. How σ maps other variables is irrelevant. For example, $\{$`x` $\leftarrow$ `(rest x)`, `a` $\leftarrow$ `(cons (first x) a)`$\}$ is a `rest`-nest substitution on `x`.

Warning: Remember that a substitution may only substitute for *variables*! While $\{$`x` $\leftarrow$ `(rest x)`, `a` $\leftarrow$ `(cons (first x) a)`$\}$ is a perfectly good `rest`-nest substitution on `x`, $\{$`x` $\leftarrow$ `(rest x)`, `nil` $\leftarrow$ `(cons (first x) nil)`$\}$ isn't even a substitution!

Rule of Inference Structural Induction ("Struct Ind"):

Summary: This rule allows you to prove ψ for all values of a variable v.

Description: If you can carry out all of the following steps:

1. select a variable v that occurs in ψ
2. let $\sigma_1, \ldots, \sigma_k$ be `rest`-nest substitutions on v
3. prove the *Base Case*:
   `(endp` v`)` $\rightarrow$ ψ
4. prove the *Induction Step*:
   `(`$\neg$`(endp` v`)` $\wedge$ ψ/σ_1 $\wedge$ ... $\wedge$ ψ/σ_k`)` $\rightarrow$ ψ

Then $\psi \leftrightarrow True$.

End of Rule

Here is an inductive proof that `(f x)`, as defined above, returns `t`.

Theorem: `(f x)` = `t`.

Proof:
Struct ind on `x`. (So let v be `x`. Let k be 1. Let σ_1 be $\{$`x` $\leftarrow$ `(rest x)`$\}$.)

Base Case: `(endp x)`:

```
lhs
=
(f x)
=                                                  {f}
(if (endp x)
    t
    (f (rest x)))
=                             {Hyp Base Case and if-t}
t = rhs.
```

Induction Step `(¬(endp x))`:
Given:
IH1: `(f (rest x)) = t` {IH stands for *Induction Hypothesis*}

```
lhs
=
(f x)
=                                                          {f}
(if (endp x)
    t
    (f (rest x)))
=                                        {Hyp Step and if-nil}
(f (rest x))
=                                                   {Hyp IH1}
t = rhs.
```

□

In this proof, IH1 is "`(f (rest x)) = t`," because it is ψ/σ_1, where ψ is "`(f x) = t`" and σ_1 maps `x` to `(rest x)`.

Note that you get one induction hypothesis for each σ_i and you can assume as many as you'd like, as long as they're all `rest`-nest substitutions on your chosen variable.

5.2 Advice About Induction (Assignment 13 cont'd)

When you write a proof by induction, *always* name the variable you're inducting on and *always* write down all of the induction hypotheses in full!

Let me illustrate the most common mistake students make when they do induction. I will do it by "proving" a non-theorem. In this example, I'll first try to prove a certain non-theorem, fail, modify my non-theorem to *another* non-theorem, and then "prove" it. I'll point out where I make the mistake in my second proof attempt. In the meantime, use this example to see how induction works.

Recall the definition of `app`.

```
(defun app (x y)
  (if (endp x)
      y
      (cons (first x)
            (app (rest x) y))))
```

Note that `(app 45 nil)` is `nil`. So ψ below is definitely not a theorem. But I will try to prove it.

ψ:
(app x nil) = x

We'll induct on x to "unwind" the recursive function app. That means we let σ_1 be {x ← (rest x)}.

The Base Case is:

(endp x) → ((app x nil) = x)

The Induction Step is

((¬ (endp x)) ∧ ((app (rest x) nil) = (rest x))) → ((app x nil) = x).

It's always a good idea to work on the Base Case first because it's usually simplest. So, expand the (app x nil) in the conclusion of the Base Case, use the (endp x) hypothesis, and rewrite with if-t:

Base Case'
(endp x) → (nil = x)

Can we prove this? Clearly not! A counterexample to this goal is x = 45, because (endp 45) is true but nil ≠ 45.

So much for that attempt! Finding a counterexample to either the Base Case or the Induction Step means that the formula you're trying to prove isn't a theorem.

Learning from this failed proof attempt, we think to ourselves: "Ok, nil is not a right identity for app when x is empty! Let's make a new conjecture: nil is a right identity for app provided x is non-empty."

ψ:
(¬(endp x)) → ((app x nil) = x).

Now I'll "prove" that. Again, we'll induct on x with the same σ_1. Now the Base Case, (endp x) → ψ, is trivial:

(endp x) → ((¬(endp x)) → ((app x nil) = x)),

because it becomes

((endp x) ∧ (¬(endp x))) → ((app x nil) = x),

which is a tautology because the hypotheses contradict one another.

So onto the Induction Step and the *fatal mistake* many students make. The correct induction step is to prove ((¬(endp x)) ∧ ψ/σ_1) → ψ. But many students set it up incorrectly, this way:

Bogus Induction Step:

```
((¬(endp x))
 ∧
 ((app (rest x) nil) = (rest x)))                {Bad IH}
→
((app x nil) = x).
```

We can prove this by expanding the `(app x nil)` in the conclusion and using the Induction Hypothesis, labeled above as *Bad IH*. The proof below is perfectly correct; the fatal mistake was setting up the wrong formula to prove. Check every step to convince yourself we really can prove *Bogus Induction Step*.

Bogus Induction Step Proof:

```
    ((¬(endp x))
     ∧
     ((app (rest x) nil) = (rest x)))
  →
    ((app x nil) = x).

↔                                                {app-of-non-endp}

    ((¬(endp x))
     ∧
     ((app (rest x) nil) = (rest x)))
  →
    ((cons (first x) (app (rest x) nil)) = x).

↔                                                {Hyp}

    ((¬(endp x))
     ∧
     ((app (rest x) nil) = (rest x)))
  →
    ((cons (first x) (rest x)) = x).

↔                                                {cons-first-rest}

    ((¬(endp x))
     ∧
     ((app (rest x) nil) = (rest x)))
  →
    (x = x).

↔                           {Equal Reflexivity and Short-Ciruit}
```

True

Since we proved the Base Case and the (Bogus) Induction Step, it seems we proved

ψ:
```
(¬(endp x)) → ((app x nil) = x).
```

But surprise! This ψ is not a theorem either! Test it for `x` = `(cons 1 45)`. The hypothesis is true, because `(cons 1 45)` is a cons and hence not empty. But `(app (cons 1 45) nil)` = `(cons 1 nil)`, which is different from `(cons 1 45)`.

So how did we prove a non-theorem? We screwed up the Induction Step. Here is the correct setup of the induction step for ψ:

Correct Induction Step:
```
 (¬(endp x))
 ∧
 ((¬(endp (rest x))) → ((app (rest x) nil) = (rest x)))        {Correct IH}
→
 ((¬(endp x)) → ((app x nil) = x)).
```

Note the difference between *Bad IH* and the correct *Correct IH*. The Induction Hypotheses are supposed to be the σ_i instances of the *entire* formula being proved. But in the Bogus Induction Step, we only wrote down the instance of the *conclusion* of our goal. We should have written down ψ/σ_1, where

ψ:
```
(¬(endp x)) → ((app x nil) = x).
```

instead of `((app x nil) = x)`$/\sigma_1$.

• **Question 222**: Can you prove the Correct Induction Step? If so, do it. If not, why not?
•

This mistake can be avoided if you follow the advice I gave at the beginning of the section: "When you write a proof by induction, *always* name the variable you're inducting on and *always* write down all of the induction hypotheses in full!" The induction step is always

`(¬(endp x))` $\wedge\ \psi/\sigma_1\ \ldots\ \ \psi/\sigma_k\ \rightarrow\ \psi$.

Don't forget that. Remember, each Induction Hypothesis is an instance of the entire formula ψ, not some part of it! If you forget this you can "prove" non-theorems.

Almost always in this course you'll have only one induction hypothesis (i.e., σ_1 will be your only substitution). So the typical induction step for $\psi_h \rightarrow \psi_c$ when inducting on `x` will be:

γ:
```
  ((¬(endp x))
   ∧
   (ψh/σ1 → ψc/σ1))
→
  (ψh → ψc).
```

Almost always, the way these proofs go is that you promote ψ_h:

$$\gamma': \quad ((\neg(\texttt{endp x})) \wedge (\psi_h/\sigma_1 \rightarrow \psi_c/\sigma_1) \wedge \psi_h) \rightarrow \psi_c.$$

and then use it to establish ψ_h/σ_1, which allows you to "detach" the conclusion of your induction hypothesis, ψ_c/σ_1, and then you use *that* to prove ψ_c. That is, if you can prove

$$(\neg(\texttt{endp x})) \wedge \psi_h \rightarrow \psi_h/\sigma_1$$

then you can convert γ' to

$$\gamma'': \quad (\neg(\texttt{endp x})) \wedge \psi_c/\sigma_1 \wedge \psi_h \rightarrow \psi_c. \qquad \{\text{detached conclusion of IH}\}$$

(The $\{\text{detached conclusion of IH}\}$ comment is printed beside the ψ_c/σ_1 line.)

At that point, and not before, you have ψ_c/σ_1 available to use in ψ_c.

A handy way to think of it is that you have to prove

hyp→hyp: $((\neg(\texttt{endp x})) \wedge \psi_h) \rightarrow \psi_h/\sigma_1$

and

concl→concl: $((\neg(\texttt{endp x})) \wedge \psi_c/\sigma_1) \rightarrow \psi_c$.

Note the symmetry: you prove an implication between the hypotheses and an implication between the conclusions, but the σ_1 is on opposite sides in the two formulas. Most students forget to prove the "hyp→hyp" part. As we saw, that is unsound.

Here is some heuristic advice about inductions. When doing an inductive proof, identify some function, f, in ψ that you want to "unwind." We picked `app` above, because it was the only function. The function you choose should be "controlled" by the variable, v, that

you choose. We picked `x` for v. Then provide as many induction hypotheses as there are different recursive calls of that function in its definition. In almost all of our problems there will only be one recursive call, on `(rest` v`)`. So you'll generally want only one substitution mapping v to `(rest` v`)`.

Your substitution can map other variables to any values you want. That will come in handy when your f has other arguments. For example, if the recursive call from `(`f u v`)` is `(`f `(+ 1` u`) (rest` v`))`, then your substitution should be $\{u \leftarrow$ `(+ 1` u`)`$, v \leftarrow$ `(rest` v`)`$\}$. Of course, you can do this only if u is a variable different from v. This may force you to change the formula you're trying to prove. Your strategy in choosing the substitution is simple: you want ψ/σ_1 to provide information about the recursive call of f. So you want σ_1 to do the same things to the arguments of f as the recursion does. If you have more than one recursive call, you'll need more than one σ_i.

By doing this, you will provide yourself with an induction hypothesis about every recursive call of the function you're unwinding. Then, in the Base Case and the Induction Step, unwind your function, use the hypotheses controlling the function to simplify what you get, and then use your induction hypotheses.

5.3 Practice Proofs (Assignment 13 cont'd)

At the end of this section are some recursive definitions. Some are very interesting programs and the theorems in the questions below state some very important properties of these programs. You might learn some valuable lessons about programming by studying these definitions and playing with them in ACL2.

You should read every question and look at every definition, even the ones not among the assigned homework problems!

When I state a question as a formula, I mean for you to prove it or exhibit a counterexample and/or countermodel.

To prove some of these theorems you will have to prove some lemmas. You can use any previously stated theorem, even if you didn't prove it. Sometimes these previously stated lemmas are crucial. Thus, it is *really important* for you to at least *read* each question because they might give you valuable hints for how to attack something I've asked you to prove.

• **Question 223**: (`listcopy-id`)

```
(list-copy x) = x
```

The relevant definition is below. •

• **Question 224**: (`alternative-true-listp`)

```
(true-listp x) = (not (terminal-marker x))
```

The relevant definition is below. •

• **QUESTION 225**: (`assoc-of-app`) Fill in the eight blanks below to make this a proof. Just provide a list of the numbers 1–8 and write the appropriate contents for each blank. Make sure your terms are all perfectly formed! Note that some of the blanks appear multiple times; e.g., the blank labelled "2" appears twice. The same thing goes in each time a blank is duplicated. The relevant definition is below.

Theorem `assoc-of-app`:
`(app (app a b) c) = (app a (app b c))`

Proof
`Induct on ____1____.`

Base Case
`(endp a) → (____2____).`

↔ {`app-identity` (page 100), twice}

`(endp a) → ((app ____3____ c) = ____4____)`

↔ {Taut}

True

Induction Step

```
  ((¬ (endp a))
   ∧
   (____5____))
 →
   (____2____)
```

↔ {`app-of-non-endp` (page 160)}

```
  ((¬ (endp a))
   ∧
   (____5____))
 →
   ((app ____6____ c)
    =
    (app a (app b c)))
```

↔ {`app-cons` (page 160)}

```
  ((¬ (endp a))
   ∧
   (____5____))
 →
   (____7____
    =
```

```
   (app a (app b c)))
↔                                        {app-of-non-endp (page 160)}
  ((¬ (endp a))
   ∧
   (____5____))
 →
   (____7____
    =
    ____8____)
↔                                        {Hyp}
  ((¬ (endp a))
   ∧
   (____5____))
 →
   (____8____
    =
    ____8____)
↔                                        {Taut}
```

True

□ •

• **QUESTION 226**: (app-right-id)
Fill in the five blanks below to make this a proof. Just list the numbers 1–5 and write the contents of the corresponding blanks. Make sure your syntax is correct!

Theorem `app-right-id:`
`(true-listp a)` → `((app a nil) = a)`

Proof Induct on ____1____.

Base Case

____2____

↔ {____3____ (see hint below)}

True

Induction Step

____4____

↔ {____5____ (see hint below)}

True

□

The relevant definition is below. **Hint**: Consider the previously proved lemmas in Questions 215 (page 162) and 216 (page 164). •

• **Question 227**: (`mapnil-app`)

`(mapnil (app a b))` $=$ `(app (mapnil a) (mapnil b))`

The relevant definitions are below. •

• **Question 228**: (`mapnil-rev`)

`(rev (mapnil a))` $=$ `(mapnil (rev a))`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **Question 229**: (`rev-app`)

`(rev (app a b))` $=$ `(app (rev b) (rev a))`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **QUESTION 230**: (`rev-rev`)

`(true-listp a)` $\to$ `((rev (rev a))` $=$ `a)`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **Question 231**: (`mem-app`)

`(mem e (app a b))` $\leftrightarrow$ `((mem e a)` $\vee$ `(mem e b))`

The relevant definitions are below. •

The amount of consing a function does is very important to efficiency. `Cons`, like `new` in Java, is expensive because it allocates new memory.

• **Question 232**: Study the function `app` (see the definition at the end of this section). How many times is `cons` called to evaluate `(app '(4 3 2) '(1))`? More generally, how many times is `cons` called to evaluate `(app a b)` when the length of `a` is n? •

• **Question 233**: Study the function `rev` (see the definitions at the end of this section). How many times is `cons` called to evaluate `(rev '(1 2 3 4))`? I want you to include the cons created explicitly in the body of `rev` and also the conses created by the call of `app`, since they're done on behalf of the `rev`. More generally, how many times is `cons` called to evaluate `(rev a)` when the length of `a` is n?

• **Question 234**: Study the function `rev1` (see the definitions at the end of this section). How many times is `cons` called to evaluate `(rev1 '(1 2 3 4) nil)`? More generally, how

many times is `cons` called to evaluate `(rev1 a nil)` when the length of `a` is n? •

Note that by comparing the answers to Questions 233 and 234 we see that `rev1` is more efficient in terms of the amount of consing it does.

`Rev1` has another interesting property. It can be executed without using much space on the "call stack." This is because `rev1` is *tail-recursive* : the result of every recursive call is returned as the result of the main call.

To understand tail-recursion it is nice to know what a "call stack" is. A "call stack" is a special case of a "push down stack." Imagine a pile of boxes, where each box is called a "frame." If you have a lot of tasks to do to complete a chore, you might write them down and put them in a box. If while doing one of the tasks, you spawn more things to do, you could write them down and put them in another box. Then you could stack the new box on top of the old. This is called "pushing" a new box or frame. You always pay attention to the top-most box. When you've done all the tasks in the top box, you "pop" by throwing away the now empty top box. What will you find? The list of things to do to complete the chore that spawned the one you just finished. So a stack is an excellent way to organize your work.

Now consider `rev`. Every time the recursive branch in `rev` is taken by the machine executing `(rev x)`, the machine must compute the values, v_1 and v_2, of `(rev (rest x))` and `(cons (first x) nil)`. Then, when those computations are done, it must remember to compute the value of applying `app` to v_1 and v_2 to get the value v that it will return. To do this, it uses a call stack. To compute `(rev x)`, the machine uses the top-most box on this pile to remember whatever intermediate results it has to store. Its "notes" are the program, the "program counter" which records where it is right now, and the values of its local variables, including spaces to hold the intermediate and final results, v_1, v_2, and v. Upon arriving at the recursive call, `(rev (rest x))`, it pushes a new box and starts over, with a new value for `x` and new spaces for v_1, v_2, and v. It uses that box until it reaches the end of that computation. Then it takes the value computed, v, out of top-most box, pops the now empty top-most box, and records that returned value into v_1 in the newly exposed top-most box. Recall that v_1 is the space to hold the value of (rev (rest x)). Note that it is now poised to continue, by computing the value of v_2, `(cons (first x) nil)`, and then `(app` v_1 v_2`)`.

It is not necessary to use a push down stack to compute `(rev1 x nil)`! You can think of this expression as being computed like this:

```
a = nil;
while ((endp x) = nil){
  a = (cons (first x) a);
  x = (rest x);
}
return a;
```

• **QUESTION 235**: `(rev1-rev-general)`

`(rev1 x a) = (app (rev x) a)`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **Question 236**: (`true-listp-rev`)

`(true-listp (rev x))`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **QUESTION 237**: (`rev1-rev`)

`(rev1 x nil) = (rev x)`

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **Question 238**: Pretend you hadn't proved `rev1-rev-general`. Try to prove `rev1-rev` directly. Why can't you? It is often hard to reason about accumulator-using functions because of this. If you want to reason about such a function, what must you do? **Hint**: What is the difference between `rev1-rev-general` and `rev1-rev`, as far as the accumulator is concerned? Why is `rev1-rev-general` easy to prove? •

• **Question 239**: (`app-rev-nil`)

`(app (rev x) nil) = (rev x)`

The relevant definitions are below. **Hint**: Do not use induction! Think about why this is true and write down your reasoning.

• **Question 240**: (`ordp-insert`)

`(ordp x)` $\rightarrow$ `(ordp (insert e x))`

The relevant definitions are below. Note that this theorem establishes that my definition of `insert` returns an ordered list if given one. You may assume the lemma that `lexorder` is a "total order," i.e., that `(lexorder a b)` $\vee$ `(lexorder b a)`. •

• **Question 241**: (`ordp-isort`)

`(ordp (isort x))`

The relevant definitions are below. Note that this theorem establishes that my insertion sort function returns an ordered list. **Hint**: Use the lemma in the previous question in your proof. •

• **Question 242**: Suppose you're a software vendor and a customer says he wants a "sorting program." You insist that he be more specific and he replies "I want you to deliver a function,

isort, with the property formalized in Question 241." How could you make a quick buck? Exhibit a trivial definition of isort that would satisfy the customer's specification without actually satisfying the cutomer's real requirements. The moral of this question is that the formal specification of software requires clear thinking. •

• **Question 243**: (isort-id)

```
(ordp x) → ((isort x) = x)
```

The relevant definitions are below. •

• **Question 244**: (isort-isort)

```
(isort (isort x)) = (isort x)
```

The relevant definitions are below. **Hint**: Some previously stated theorems may be of help. •

• **Question 245**: (perm-reflexive)

```
(perm x x)
```

The relevant definitions are below. You may assume without proof the following trivial lemmas. You ought to be able to prove them.

```
mem-first:
(¬(endp x)) → (mem (first x) x)
del1-first:
(¬(endp x)) → ((del1 (first x) x) = (rest x))  •
```

• **Question 246**: (perm-symmetric)

```
(perm x y) → (perm y x)
```

The relevant definitions are below. You may assume without proof the following lemmas.

```
endp-perm:
(endp x) → ((perm x y) = (endp y))

not-endp-perm:
  (¬(endp x))
→
  ((perm x y)
   ↔
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y))))

mem-implies-endp:
```

```
(mem e x) → (¬(endp x))
```

```
perm-del1-rest:
  ((¬(endp x))
   ∧
   (mem (first x) y))
→
  ((perm (del1 (first x) y) (rest x))
   ↔
   (perm y x))
```

The first three are easy to prove. The last is an interesting challenge but you don't have to prove it to use it in this question. •

• **Question 247**:

```
((perm x y) ∧ (perm y z)) → (perm x z)
```

The relevant definitions are below. You may assume without proof the following lemmas.

```
endp-perm:
(endp x) → ((perm x y) = (endp y))
```

```
not-endp-perm:
  (¬(endp x))
→
  ((perm x y)
   ↔
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y))))
```

```
perm-mem:
((perm x y) ∧ (mem a x)) → (mem a y)
```

```
perm-del1:
(perm x y) → (perm (del1 a x) (del1 a y))
```

The last two are interesting challenges but you may use them here without proving them. •

• **Question 248**: (perm-cons)

```
(perm x y) → (perm (cons e x) (cons e y))
```

The relevant definitions are below. •

• **Question 249**: (perm-app1)

```
(perm x y) → (perm (app x z) (app y z))
```

The relevant definitions are below. •

• **Question 250**: (`perm-app2`) Prove

```
(perm x y) → (perm (app z x) (app z y))
```

The relevant definitions are below. •

• **Question 251**: (`perm-rev`)

```
(perm x y) → (perm (rev x) (rev y))
```

The relevant definitions are below. •

• **Question 252**: (`perm-rev-id`)

```
(perm (rev x) x)
```

The relevant definitions are below. •

• **Question 253**: (`perm-isort`)

```
(perm (isort x) x)
```

The relevant definitions are below. •

• **Question 254**: There are, of course, many ways to sort a list. Insertion sort (`isort`) successively inserts each element at the right spot in the evolving answer. Merge sort splits the list into two partitions, e.g., the elements that occur in the even numbered positions and the elements that occur in the odd. Then it sorts each list. Finally, it merges the two ordered list, "shuffling" them together. Define `msort` to be a merge sort function. •

• **Question 255**: Suppose you've proved that both `isort` and `msort` are correct in the sense that each returns an ordered permutation of its input. That is, you may assume

```
ordp-isort:  (ordp (isort x))
perm-isort:  (perm (isort x) x)
ordp-msort:  (ordp (msort x))
perm-msort:  (perm (msort x) x)
```

Prove `(equal (isort x) (msort x))` from these assumptions together with any other valid lemmas you'd like to use. You will not need definitions for `isort` and `msort` to answer this question. You do not need to prove the lemmas you invent for this proof, just make sure every lemma you assume is always true! **Hints**: There is only one ordered permutation of a list. Formalize that lemma and use it together with the assumptions above and the lemmas of Questions 246 and 247 stating that `perm` relationship is symmetric and transitive. •

Here are the relevant definitions, listed alphabetically.

```
(defun app (x y)
  (if (endp x)
      y
      (cons (first x)
            (app (rest x) y))))


(defun del1 (x y)
  (if (endp y)
      y
      (if (equal x (first y))
          (rest y)
          (cons (first y)
                (del1 x (rest y))))))


(defun insert (e x)
  (if (endp x)
      (cons e x)
      (if (lexorder e (first x))
          (cons e x)
          (cons (first x)
                (insert e (rest x))))))


(defun isort (x)
  (if (endp x)
      nil
      (insert (first x)
              (isort (rest x)))))


(defun list-copy (x)
  (if (endp x)
      x
      (cons (first x)
            (list-copy (rest x)))))


(defun mapnil (x)
  (if (endp x)
      nil
      (cons nil (mapnil (rest x)))))
```

```
(defun mem (x y)
  (if (endp y)
      nil
      (if (equal x (first y))
          t
          (mem x (rest y)))))


(defun ordp (x)
  (if (endp x)
      t
      (if (endp (rest x))
          t
          (and (lexorder (first x) (first (rest x)))
               (ordp (rest x))))))


(defun perm (x y)
  (if (endp x)
      (endp y)
      (and (mem (first x) y)
           (perm (rest x)
                 (del1 (first x) y)))))


(defun rev (x)
  (if (endp x)
      nil
      (app (rev (rest x))
           (cons (first x) nil))))


(defun rev1 (x a)
  (if (consp x)
      (rev1 (rest x) (cons (first x) a))
      a))


(defun terminal-marker (x)
  (if (endp x)
      x
      (terminal-marker (rest x))))


(defun true-listp (x)
```

```
(if (consp x)
    (true-listp (rest x))
    (equal x nil)))
```

6

The Elementary Arithmetic Waiver (Assignment 14: 7 days)

I have chosen not to distract you with the axioms for arithmetic. Just trust me: elementary arithmetic can be formalized in ways analogous to our formalization of lists.

In proofs for this course: you may assume without proof any theorem you want about ACL2 arithmetic provided

1. you state the theorem as a formula,
2. the formula mentions no function symbols other than: addition (`+`), difference (`-`), product (`*`), rational division (`/`), exponentiation (`expt`), the modular arithmetic functions `floor` and `mod`, equality (`equal`), the standard inequalities (`<`, `<=`, `>=`, and `>`), the recognizers `natp`, `integerp`, and `zp`, the function `nfix` for casting into the naturals (page 57), the propositional functions `and`, `or`, `not`, `implies`, and `iff` (possibly written as $\wedge$, $\vee$, $\neg$, $\rightarrow$, and $\leftrightarrow$), and numeric constants.
3. the formula you write really is always true of the ACL2 functions in it!

Because the ACL2 logic is untyped, the ACL2 arithmetic functions have some strange properties: e.g., `(+ nil 6)` is `6` and `(/ 25 0)` is `0`. It is always a good idea to err on the safe side and add hypotheses about the variables in your "theorems" to insure they're integers or appropriate naturals.

7

Quantifiers (Assignment 14 cont'd)

Traditional first order logics provide two features that ACL2 does not provide directly, namely "universal" and "existential" quantifiers. These are traditionally written as an upside-down "A" ($\forall$) and a backwards "E" ($\exists$). You will see these symbols often in your CS education. So I'll teach you the rudiments here.

However, if you are asked to prove a statement of the form $(\forall x \forall y \ldots : \psi)$, where all of the quantifiers are "$\forall$" and ψ is an ACL2 formula (and thus has no quantifiers in it) then you can prove the quantified formula by proving ψ the way we've been doing. That is, if ψ is a theorem, then $(\forall x \forall y \ldots : \psi)$ is a theorem.

So, when you proved:

```
(app (app x y) z) = (app x (app y z))
```

you also proved

```
(∀ x ∀ y ∀ z :  (app (app x y) z) = (app x (app y z))).
```

But beware! ψ and $(\forall x \forall y \ldots : \psi)$ are not equivalent: You can't just replace any occurrence of one by the other.

In a sense, to prove any quantified formula you reduce it to a different formula without quantifiers and then you prove that formula using the techniques we've already studied. The question is "how do you get rid of the quantifiers?"

But we're getting ahead of ourselves because you probably don't know what "$\forall$" and "$\exists$" mean. So we'll start at the beginning now.

7.1 The Universal Quantifier (Assignment 14 cont'd)

If ϕ is a formula and v is a variable symbol, then in standard first order logic, $(\forall v : \phi)$ is also a formula. This is is read as "for all v, it is always the case that ϕ" or simply "for all v, ϕ".

For example, since this is a formula:

```
(how-many e x) = (how-many e y)
```

then, in standard first order logic, so is this:

```
(∀ e :  (how-many e x) = (how-many e y))
```

In some books, instead of "$(\forall v : \phi)$," you'll see variations like

$(\forall v \phi)$
$(\forall (v) \phi)$
$(\forall v . \phi)$
$(\forall v) \phi$

and other variations.

Regardless of the notation, there is the *quantifier* symbol (the "for all" sign ("$\forall$")), a *bound variable* (v), and a formula, ϕ, often called the *matrix*, but here called the *body* of the quantified formula.

The bound variable, v, is understood to range over some fixed non-empty universe of objects, typically called the *universe* (or "domain") and denoted by $\mathcal{D}$. $\mathcal{D}$ contains all the objects we can "talk about" in the logic. So it is not unusual to see people say, "suppose the universe contains only the natural numbers," or "The domain is the set of real numbers, Booleans, and ordered pairs constructed from elements of the domain." Once a domain is designated or understood, the variable symbols in formulas are understood to take on values from $\mathcal{D}$. Since $\mathcal{D}$ is non-empty, at least we know our variables denote *something*. The idea is that "$(\forall v : \phi)$" means that ϕ holds for every value of v chosen from $\mathcal{D}$.

The ACL2 universe contains all the natural numbers, rationals, and complex rationals, all the character objects, all the strings, all the symbols, and all the lists (conses) you can build from elements of that universe. In ACL2, the universe is infinite. But whether the universe is infinite or finite, the basic laws for manipulating quantifiers will be unaffected.

So what does "$(\forall v : \phi)$" mean? Questions of this kind often stump new CS students, because it's hard to come to grips with the meaning of the verb "to mean." But in the context of logic, the *meaning* of a formula is synonymous with its *truth value*. So let me rephrase the question: given a particular v and ϕ, how do we determine whether "$(\forall v : \phi)$" is $True$ or $False$? This should at least be easy for you to think about because it's equivalent to: how do you evaluate "$(\forall v : \phi)$"?

To evaluate "$(\forall v : \phi)$," map over $\mathcal{D}$ and for each element, d, bind v to d and evaluate ϕ. If the result is $False$, then the value of "$(\forall v : \phi)$" is $False$. Otherwise, keep going. If ϕ returns $True$ for every d in $\mathcal{D}$, then the value of "$(\forall v : \phi)$" is $True$.

Of course you see the problem with this description of a "computation:" if $\mathcal{D}$ is infinite this "algorithm" will never terminate. There's nothing we can do about that! To understand "$(\forall v : \phi)$" you just have to be able to *imagine* evaluating ϕ on a possibly infinite number of different values of v. Mathematicians do it all the time. Get used to it.

We can make quantification very concrete if we focus on a finite domain and pick a particular

ϕ. Imagine for a moment that $\mathcal{D}$ is the list `(0 1 2 3)` and let's consider

```
(∀ e :  (how-many e x) = (how-many e y))
```

Then the meaning of this quantified statement is just:

```
( ((how-many 0 x) = (how-many 0 y))
  ∧
  ((how-many 1 x) = (how-many 1 y))
  ∧
  ((how-many 2 x) = (how-many 2 y))
  ∧
  ((how-many 3 x) = (how-many 3 y)))
```

You may think of the universal quantifier as a big `and` ($\wedge$) with one conjunct for each element of $\mathcal{D}$.

If the universe can be expressed as a list in ACL2, then we could write down an ACL2 expression that is equivalent to

$(\forall\ v\ :\ \ \phi)$

as follows:

- let $x_1, \ldots, x_n$ be the variables in ϕ other than v (actually, we want $x_1, \ldots, x_n$ to be the "free" variables in ϕ, but I'll explain below)
- make up a new function symbol, f,
- define

  ```
  (defun f (lst x1 ... xn)
    (if (endp lst)
        t
        (and (let ((v (first lst))) φ)
             (f (rest lst) x1 ... xn))))
  ```

Then $(\forall\ v\ :\ \phi)$ is equivalent to $(f\ \mathcal{D}\ x_1 \ldots x_n)$.

So if ϕ is `((how-many e x) = (how-many e y))` and we let f be the symbol `foo`, then

```
(∀ e :  (how-many e x) = (how-many e y))
```

is just `(foo` $\mathcal{D}$ `x y)`, where

```
(defun foo (lst x y)
  (if (endp lst)
      t
```

```
      (and (let ((e (first lst)))
             (equal (how-many e x) (how-many e y)))
           (foo (rest lst) x y))))
```

Suppose I asked the question: "Write an ACL2 expression that determines, for given values of `x` and `y`, whether `((how-many e x) = (how-many e y))` is true for all values of `v` in `D`. You are free to define a function if you wish." What would you have done? I hope you would have defined a version of `foo` above and answered the question with "`(foo D x y)`."

• **Question 256**: Write an ACL2 expression that determines whether every element in `D` is a natural number. You are free to define a function if you wish. If you answer correctly, that expression is the meaning of "∀`v` : `(natp v)`," if the domain $\mathcal{D}$ were `D`. •

• **Question 257**: Write an ACL2 expression that determines whether every element in `D` is a natural number larger than `x`, for some given value of `x`. You are free to define a function if you wish. •

• **Question 258**: Assuming $\mathcal{D}$ is `D`, write the quantified expression corresponding to Question 257, page 194. See section 7.3 (page 198) for instructions on how to denote quantifiers in your typed homework. •

The biggest trap in dealing with quantifiers is handling the notion that v is "bound" by the expression.

You know we can use the hypothesis below to transform

```
(x = 3) → (A x (p x))
```

to any of the following:

```
(x = 3) → (A 3 (p x))
(x = 3) → (A x (p 3))
(x = 3) → (A 3 (p 3)).
```

That's just replacement of equals by equals using our Hypothesis rule.

But now imagine "turning the `A` upside down" and see how different the situation is. Suppose $\mathcal{D}$ contains the natural numbers. I hope you see that it would be crazy to transform

```
(x = 3) → (∀ x :  (p x))
```

to

```
(x = 3) → (∀ 3 :  (p x))
```

or to

```
(x = 3) → (∀ x :  (p 3))
```

or to

`((x = 3) → (∀ 3 : (p 3)).`

The first and third transformations simply make no sense at all. How can we imagine letting `3` take on different values from $\mathcal{D}$? Whatever follows the "∀" must be a variable symbol! The second transformation radically changes the meaning (value) of the formula. For example, this formula always evaluates to true:

`(x = 3) → (∀ x : ((x = 2) → (x ≠ 3))).`

Why? The conclusion is true. For every `x` in $\mathcal{D}$, if `x` is `2` then obviously it is not `3`. When the conclusion of an implication is true, the implication is true, no matter what the hypothesis is.

However, this formula:

`(x = 3) → (∀ x : ((x = 2) → (3 ≠ 3))).`

is false. Why? Because the body of the quantified expression is false for at least one of the values for `x` chosen from $\mathcal{D}$, namely `2`.

So if we transformed

`(x = 3) → (∀ x : ((x = 2) → (x ≠ 3)))`

to

`(x = 3) → (∀ x : ((x = 2) → (3 ≠ 3)))`

"by replacement of equals by equals" we would be making a real mistake! Something is wrong with our interpretation of "replacement of equals by equals" in the context of quantified expressions.

Those occurrences of `x` in the conclusion are very different from the usual occurrences of `x`: they are "bound" by the quantifier. We define those concepts formally in Section 7.4. But you can think of the bound variables like locals of a method, invisible to the outside.

There is a similar problem about using previously proved theorems to rewrite our goal. We know `(¬(consp x)) → ((first x) = nil)` is a theorem (actually, an axiom). So can we use the Rewrite rule of inference (page 128) to transform

`(¬(consp x)) → (∀ x : (first x) = nil)`

to

`(¬(consp x)) → (∀ x : nil = nil)`?

Clearly not. The problem is with relieving the hypothesis of the axiom. We know `(¬(consp x))` from our goal and according to step 6 of the Rewrite rule of inference, we have to prove `(¬(consp x))`. But the `x` in the first one represents a different thing from the `x` in the

second! We are going to severely restrict the Rewrite rule of inference when rewriting inside quantified expressions.

Warning: Just remember, you can't substitute equals for equals inside quantifiers without checking some extra conditions!

I bet that most of the mistakes you'll make when dealing with quantifiers will have to do with bound variables. Be careful and pay attention! This kind of trap is just one of many we'll discuss later.

We now have two classes of variables: those that are "bound" and those that are "free."

For example, suppose `p` is a predicate of arity 3 and consider `(p x y z)`. Then `x`, `y`, and `z` are free variables of `(p x y z)`. But only `x` and `z` are free variables in (∀`y` : `(p x y z)`). To make matters worse, it's really not *variables* that are free or bound but *occurrences of variables*. Of course, to talk about "occurrences" we'll resort yet again to our notion of addresses.

Consider the formula below, which I'll call γ. There are eight places that variables occur in γ and I point to each below.

```
γ:
(∀ y :  (p x  y  z)) ∧ (∀ x :  (p x  y  z))
   ↑        ↑  ↑  ↑          ↑        ↑  ↑  ↑
   π1       π2 π3 π4         π5       π6 π7 π8
```

We say the `y` at π_1 is a "binding" occurrence of `y`. The `y` at π_3 is a "bound" occurrence of `y`. The variables at π_2 and π_4 are "free" occurrences. Thus, in the first conjunct of γ, `y` occurs bound and `x` and `z` occur free. In the second conjunct, `x` is bound by the binding occurrence at π_5 and occurs bound at π_6. Both `y` and `z` occur free at π_7 and π_8. (By the way, we will never need to point to a binding occurrence and so π_1 and π_5 above are actually going to be illegal addresses.)

Thus, `x`, `y`, and `z` appear free in γ, and `x` and `y` also appear bound (in other occurrences).

7.2 The Existential Quantifier (Assignment 14 cont'd)

The existential quantifier is the dual of the universal. If you think of $(\forall v : \phi)$ as a big `and` $(\wedge)$ with one conjunct for each element of $\mathcal{D}$, then think of $(\exists v : \phi)$ as a big `or` $(\vee)$ with one disjunct for each element of $\mathcal{D}$.

We needn't explain the existential quantifier at all: many logic books just define existential quantification in terms of universal:

$$(\exists v : \phi) \leftrightarrow (\neg(\forall v : (\neg\phi))).$$

But it can be awkward to work with "∃" as an abbreviation for a negated universal of a

negation.

If ϕ is a formula and v is a variable symbol, then in standard first order logic, $(\exists v : \phi)$ is also a formula. This is is read as "there exists a v such that ϕ is true" or "there is a v such that ϕ."

As with the universal quantifier, you see many different notations.

$(\exists v \phi)$
$(\exists (v) \phi)$
$(\exists v . \phi)$
$(\exists v) \phi$

but in all of them we see the quantifier symbol (the "exists" sign $(\exists)$), a bound variable (v) and a body formula (ϕ). As with the universal quantifier, the bound variable ranges over $\mathcal{D}$, so that "$(\exists v : \phi)$" means that for some value of v chosen from $\mathcal{D}$, ϕ holds.

To evaluate "$(\exists v : \phi)$," map over $\mathcal{D}$ and for each element, d, bind v to d and evaluate ϕ. If the result is $True$, then the value of "$(\exists v : \phi)$" is $True$. Otherwise, keep going. If ϕ returns $False$ for every d in $\mathcal{D}$, then the value of "$(\exists v : \phi)$" is $False$.

Imagine for a moment that $\mathcal{D}$ is the list `'(0 1 2 3)` and let's consider

```
(∃ e :  (how-many e x) = (how-many e y))
```

Then the meaning of this quantified statement is just:

```
( ((how-many 0 x) = (how-many 0 y))
  ∨
  ((how-many 1 x) = (how-many 1 y))
  ∨
  ((how-many 2 x) = (how-many 2 y))
  ∨
  ((how-many 3 x) = (how-many 3 y)))
```

If the universe can be expressed as a list in ACL2, then we could write down an ACL2 expression that is equivalent to

$(\exists\ v\ :\ \ \phi)$

as follows:

- let $x_1, \ldots, x_n$ be the free variables in ϕ other than v
- make up a new function symbol, f,
- define

```
(defun f (lst x1 ... xn)
  (if (endp lst)
```

```
        nil
        (or (let ((v (first lst))) φ)
            (f (rest lst) x1 ... xn))))
```

Then $(\exists\ v\ :\ \phi)$ is equivalent to $(f\ \mathcal{D})$.

We could define the existential quantifier in terms of the universal one:

$$(\exists\ v\ :\ \phi) \leftrightarrow (\neg(\forall\ v\ :\ (\neg\phi))).$$

• **Question 259**: Write down an informal argument that the values returned by the two sides of the "↔" above are identical, given the fact that a quantified formula is a "big conjunct" and an existentially quantified formula is a "big disjunct." See section 7.3 (page 198) for instructions on how to denote quantifiers in typed homework. •

• **Question 260**: We could also define the universal quantifier in terms of the existential one. Write the formula. See section 7.3 (page 198) for instructions on how to denote quantifiers in typed homework. •

• **Question 261**: Paraphrase the following formula in English.

$(\exists\ y\ :\ (\texttt{natp}\ y) \wedge\ x * y > 0)$ •

• **Question 262**: Is the statement in the question above true for all possible values of x false for all possible falues of x, or true for some values of x and false for others? •

• **Question 263**: Paraphrase the following in English.

$(\forall\ x\ :\ (\exists\ y\ :\ (\texttt{natp}\ x) \rightarrow ((\texttt{natp}\ y) \wedge ((x + y) = 0))))$ •

• **Question 264**: Is the statement above always true, always false, or sometimes true and sometimes false? •

• **Question 265**: Write the quantified statement that means "for every x and y, there is a list that contains x as its first element and y as its second." See section 7.3 (page 198) for instructions on how to denote quantifiers in typed homework. •

7.3 ASCII Substitutes for Quantifiers (Assignment 14 cont'd)

This chapter introduces three symbols that are not in the ASCII character set, "∀", "∃", and (later in the chapter) "⊣" (sometimes called a *reversed turnstyle*). Please use these substitutions in your homeworks:

	ASCII substitute
$(\forall\ v\ :\ \psi)$	`(all` v ψ`)`
$(\exists\ v\ :\ \psi)$	`(exists` v ψ`)`
$\psi \dashv \psi'$	ψ -! ψ'

Note that I do not want you to type the colon that I use in the text. Note also that the substitute for reversed turnstyle uses an exclamation point, not a vertical bar. These conventions are necessary due to Lisp's treatment of so-called "signs" in symbol names.

Warning: If you type a colon to ACL2 and don't know what you're doing (!) it may print a scary warning including something like

```
 ************ ABORTING from raw Lisp ***********
```

Just ignore it and correct your input.

Thus, you may type

(∀ x : (∃ y : (natp x) → ((x × y)= x)))

as:

```
(all x (exists y (natp x) --> ((x * y) = x)))
```

and you may type

(∀ x : (∃ y : (x + y) = x) → (natp x))

as:

```
(all x (exists y (x + y) = 0) --> (natp x)).
```

7.4 Talking About Quantifiers Precisely (Assignment 15: 2 days)

With both quantifiers, the occurrence of v immediately after the quantifier in $(\forall v : \phi)$ and in $(\exists v : \phi)$ is a *binding* occurrence. Any occurrence of v inside ϕ is a *bound* occurrence. Any occurrence of a variable that is neither a binding occurrence nor a bound occurrence is a *free* occurrence.

A variable is *free* in a formula if there is a free occurrence of the variable. A variable is *bound* in a formula if there is a bound occurrence of the variable.

• **QUESTION 266**: Write a formula in which the variable `s` is both bound and free. •

• **Question 267**: What are the free variables in

((natp p) ∧ (∃ n : (* 2 n) = p))? •

A formula is *closed* if it contains no free variables. In a closed formula is also said to be a *sentence*, but we won't use that terminology here. The *universal closure* of a formula ψ containing the free variables $v_1, v_2, \ldots, v_n$, listed in left-to-right order of appearance, is $(\forall v_1 : (\forall v_2 : \ldots (\forall v_k : \psi) \ldots))$.

A variable v is in the *bound variable list* for an address π in ψ if a bound occurrence of v would be created by replacing the subformula at π in ψ by v.

For example, let ψ be `(p z)` $\wedge$ ($\forall$`x` : `(q z)` $\wedge$ ($\exists$`y` : `(r z)`)). At the address that points to `(p z)`, the bound variable list is empty: the bound variable list contains only symbols that would appear bound if put at that address and there are no such symbols. At the address that points to `(q z)`, the bound variable list is just `(x)` because if we put an `x` at that address, it would be a bound occurrence; no other variable would be bound. At the address that points to `(r z)`, the bound variable list is `(x y)` – or `(y x)` since I haven't specified order. Another way to put it is that the bound variable list includes a variable only if the formula at π appears in the *scope* of a quantifier that binds that variable.

We say an address π in ψ is *in the scope of a quantifier* if the bound variables list at π is non-empty.

We need to update our notion what equivalence relation must hold between α and β in order to replace α at π by β in γ while maintaining either `equal` or `iff` for γ. The body of a quantifier is just tested, so propositional equivalence of the quantified formula is maintained by preserving the propositional equivalence of the body. To prevent the replacement of binding occurrences of variables, such as the `x` in ($\forall$`x` : ψ), we don't allow addresses to point to such locations (just as we don't allow them to point into constants).

• **QUESTION 268**: Suppose π is the address of the `(* p v)` in the formula below.

`((natp p) ∧ ((natp q) ∧ (divides p q))) → (∃ v : (* p v) = q)`?

What is the equivalence relation admitted by π in that formula to maintain propositional equivalence? Does π admit propositional replacement? •

• **QUESTION 269**: Let π be the address of the equality subterm in the formula above. Does π admit propositional replacement? •

We've already observed that there is a problem with "replacement of equals for equals" in the scope of a quantifier. You can't use the hypothesis below to substitute `3` for `x` in the conclusion and transform

```
(x = 3) → (∀ x : (p x))
```

to something like

```
(x = 3) → (∀ x : (p 3)).
```

The existence of free and bound variables in our formulas also raises problems with instantiation, i.e., applying substitutions.

First, when we apply a substitution we must not replace bound occurrences of variables even if they are mentioned in the substitution. If we have proved the theorem

```
γ:
(∀ x : (p x)) → (q x y)
```

and we wish to instantiate it with the substitution $\sigma = \{$`x` $\leftarrow$ `3`$\}$, we cannot just replace the bound occurrence of `x`. That is, it would be wrong to conclude from the theorem γ that

γ':
(∀ x : (p 3)) → (q 3 y)

is a theorem!

• **QUESTION 270**: Define `p` and `q` so that γ, above, is always true but so that γ' is not always true. You may assume that $\mathcal{D}$ consists of the natural numbers. That is produce a model for γ and a counterexample to γ' under that model. **Note**: You will not be able to evaluate either γ or γ' in ACL2 (since quantified formulas cannot be evaluated over infinite domains), but I recommend that you define your `p` and `q` in ACL2 and experiment with them. At least that will insure that your syntax is correct. Just turn in the two `defun`s. •

Summary Warning: You can't instantiate bound variables!

The second problem is illustrated below. It should be clear to you that the formula γ is valid, even if we don't have the machinery yet to prove it:

γ:
(∀ x : x=y) → (u = v).

Recall that $\mathcal{D}$ is non-empty. So it has one or more objects in it. If there is only one object in the universe, then the hypothesis is true and so is the conclusion: all the variables denote that one object. If there are multiple objects in the universe, then the hypothesis is false – not every object in $\mathcal{D}$ is equal to which ever object `y` denotes – so the implication holds vacuously. In both cases, γ is always true.

Since γ is valid, we ought to be able to instantiate it and obtain a valid formula. Let γ' be γ/σ, where σ is $\{$`y` $\leftarrow$ `x`$\}$. If we use our old notion of instantiation γ/σ is:

γ':
(∀ x : x=x) → (u = v).

This is clearly not valid! The hypothesis is always true – of course each element of $\mathcal{D}$ is equal to itself – and so γ' is equivalent to `u= v`. But that formula isn't valid.

What happened? How did we produce a non-theorem by instantiation of a theorem?

The answer is that the `x` in our substitution σ was *captured* by the quantifier when we plugged it in for `y`. Before the instantiation, that `y` was free – not bound by the quantifier. Afterwards, the variable at that address was bound. This is bad.

Summary Warning: You can't let your free variables be captured!

To avoid it, we *redefine* the notion of instantiation to avoid capture. Since the bound variables are "local" their names shouldn't matter and we simply rename them as necessary to avoid capturing free variables brought in by the substitution. A proper instantiation of γ by σ is

γ':
`(∀ x0 : x0=x) → u = v.`

So to apply substitution σ to ψ, you dive through ψ doing what you normally would until you encounter a quantifier. That is, when you encounter a variable, you return its image in σ (or the variable itself if it is not hit by σ). When you encounter a constant, you return the constant. When you encounter a function application, $(f\ a_i\ \ldots\ a_n)$ you apply the subsitution to each argument and return the result: $(f\ a_i/\sigma\ \ldots\ a_n/\sigma)$. But when you encounter $(\forall\ v\ :\ \beta)$ you behave very differently. I'll explain that case. The case for $(\exists\ v\ :\ \beta)$ is exactly analogous.

When you encounter $(\forall\ v\ :\ \beta)$, first, compute the free variables in that formula. Call that list Υ. (Note that v is never a member of Υ.) Υ lists the variables that the substitution can legitimately replace, so next "restrict" the substitution to those variables by deleting the bindings of all other variables. Call restricted substitution σ'.

Next, make up a new variable name, v', to be used in place of v; your v' must have two properties: (i) v' must not be among the variables in Υ and (ii) it should not be free in any of the terms in the range of σ'. If v itself has those properties, you can let v' be v.

Once you've chosen a suitable v', you will have to replace free occurrences of v in β by v'. You can do that by extending σ' to contain one more binding, namely "$v \leftarrow v'$". Let that extended σ' be σ''. Finally, to finish, return $(\forall\ v'\ :\ \beta/\sigma'')$.

The $\exists$-expression is handled analogously.

Here is an example. Let σ be the substitution {`x` ← `(f a)`, `y` ← `(g a b)`}. The result of applying σ to the formula below:

```
(p x y) ∧ (∀ x :  (r x) ∧ (∃ a :  (q x y a)))
```

is

```
(p   x     y)        ∧ (∀ x :  (r x) ∧ (∃ a   :  (q x y        a)))/σ
```

which is

```
(p (f a) (g a b)) ∧ (∀ x :  (r x) ∧ (∃ a1 :  (q x (g a b) a1)))
```

Here is the blow-by-blow. When we apply σ to the first conjunct, `(p x y)`, we have not yet encountered any quantifiers and so just apply the substitution as before. When we move over to the `(∀ x : ...)` we delete the binding "`x` ← `(f a)`" from the substitution we're using on the body of the quantifier expression. That leaves us with {`y` ← `(g a b)`}. Υ, the free variables in the body, is `(y)` because the other variables in the body, `x` and `a`, are bound. We consider renaming `x` and first just consider whether `x` itself will do for v'. (i) `x` is not in Υ and (ii) `x` is not free in any of the terms in the range of our substitution – the only free variables in the range of our substitution are `a` and `b`. So we just let v' be `x`. Applying that to the body, we first encounter the `(r x)`, which is unchanged because `x` is not hit by the modified substitution in this scope. We next encounter `(∃ a : (q x y a))`.

The Υ for this expression is `(x y)`. We restrict our substitution to those variables but that doesn't change it, since only `y` is hit. Next, we consider renaming the bound variable `a`. (i) `a` isn't a member of Υ, so it looks like a candidate, but (ii) `a` is a free variable in the range of our substitution, $\{$`y` $\leftarrow$ `(g a b)`$\}$. So we can't use `a`. We're also unable to use `x` and `y` (they're in Υ) and `b` (it is free in the range of the substitution). So we ultimately choose `a1`. We change our substitution to $\{$`y` $\leftarrow$ `(g a b)`, `a` $\leftarrow$ `a1`$\}$. Applying that to the body of the $\exists$-expression gives `(q x (g a b) a1)` and we return a new $\exists$-expression with binding occurrence `a1` and that instantiated body. And we're done!

The above description is very subtle. Even logicians who study this stuff have gotten the definition wrong. More troubling, students who try to carry out the instantiation algorithm frequently take short-cuts and make mistakes. The two common mistakes in dealing with quantifiers are

- ♦ substituting for bound variables, e.g., transforming (`x` = `3`) $\rightarrow$ ($\forall$`x` : `(p x y)`) to (`x` = `3`) $\rightarrow$ ($\forall$`x` : `(p 3 y)`)
- ♦ allowing your free variables to be captured when you instantiate, e.g., instantiating the `y` in ($\forall$`x` : `(p x y)`) with `(f x)` to erroneously get ($\forall$`x` : `(p x (f x))`) instead of the correct ($\forall$`x0` : `(p x0 (f x))`).

• **Question 271**: What is γ/σ where:

γ: `(∀ x : (p x y))`
σ: `{ x ← a, y ← b }` •

• **Question 272**: What is γ/σ where:

γ: `(∀ x : (p x y))`
σ: `{ x ← a, y ← (f x)}` •

• **Question 273**: What is γ/σ where:

γ: `(∀ x : ((p x y) ∧ (∃ y : (q x y z))))`
σ: `{ x ← a, y ← (g x), z ← (h a x)}` •

• **QUESTION 274**: What is γ/σ where:

γ: `(∀ x : (p x y)) ∧ (∀ x : (∃ y : (q x y z)))`
σ: `{ x ← a, y ← (g x), z ← (h a x)}` •

• **Question 275**: What is γ/σ where:

γ: `(∀ x : (p x y)) ∧ (∀ x : (∃ y : (q x y z)))`
σ: `{ x ← a, y ← (g x), z ← (h a y)}` •

7.5 Imprecise Talk about the Rules of Inference (Assignment 15 cont'd)

We have to add some new rules of inference to deal with quantifiers. In this section I'll introduce the new rules informally and with examples. In the next section I'll be precise.

One rule will allow us to rename the bound variables in a formula. For example, we can transform `(∃ x : (p x y))` to `(∃ z : (p z y))` without changing its meaning (value). Note however that we could not change the bound variable from `x` to `y`; `(∃ y : (p y y))` means something different.

• **Question 276**: Define a function `p` so that `(∃ x : (p x y))` is always true but `(∃ y : (p y y))` is false, assuming $\mathcal{D}$ contains the natural numbers. •

The rule that allows this is called "alpha conversion" and it corresponds exactly to the phenomenon in programming where you decide to rename a local variable – but you have to be careful that your "new" variable name is really new! You can change the meaning (value) of your program if you replace a local variable name by the name of some other variable in the same scope.

In programming, "inlining" is the idea of replacing a procedure or method call by its body to save the overhead of an invocation. When you inline, you have to be careful to avoid over-writing the variables in the caller's code. Imagine if the caller has the local variable `x` and the called procedure has a local variable `x` which it initializes to `0`. If you inlined the procedure you might smash the value of the caller's `x`. To avoid this, you might do alpha conversion on the procedure first – rename its local variable to `z`, but you must avoid using any name already in use in the procedure. In logic, we use alpha conversion in a very similar situation.

There are four main rules for manipulating quantifiers, two that deal with the universal symbol and two that deal with the existential. The reason there are two of each is that how you deal with those quantifiers depends on whether they occur in the hypothesis or the conclusion of what you're proving.

The rule for dealing with a "∀" in the hypothesis is symmetric with the rule for dealing with an "∃" in the conclusion; the rule for dealing with a "∀" in the conclusion is symmetric with the rule for dealing with an "∃" in the hypothsis.

• **Question 277**: Why should this symmetry be present? •

In all of the examples below, let's assume $\mathcal{D}$ contains the integers.

For example, suppose you're trying to prove

```
(∀ x :  (p x)) → (p 3)
```

This should feel like an obvious truth: if `p` is true for all `x`, then of course it is true for `3`. To prove such a rule, we need to be able to instantiate a universally quantified hypothesis. Before we show that, consider this formula:

$(\forall$ `x :  (p x))` $\rightarrow$ `((p 3)` $\wedge$ `(p 7)).`

It should be just-as-obviously true!

When you have a universally quantified hypothesis you can drop the quantifier and replace the bound variable by any term you want. What's more, you can have your cake and eat it too! You can keep the quantified hypothesis around in case you need it again! We call this *copying the hypothesis*.

So we will be able to transform

$(\forall$ `x :  (p x))` $\rightarrow$ `((p 3)` $\wedge$ `(p 7)).`

first into

`((p 3)` $\wedge$ $(\forall$ `x :  (p x)))` $\rightarrow$ `((p 3)` $\wedge$ `(p 7)).`

and then into

`((p 3)` $\wedge$ `(p 7)` $\wedge$ $(\forall$ `x :  (p x)))` $\rightarrow$ `((p 3)` $\wedge$ `(p 7)).`

by using this rule for dealing with a "$\forall$" in the hypothesis. Of course, once we have made the second transformation above, the goal formula is an instance of a tautology.

Earlier we talked about the symmetry of a "$\forall$" in the hypothesis with an "$\exists$" in the conclusion. Here's an example. How can we prove this?

`(p 3)` $\rightarrow$ $(\exists$ `x :  (p x))`?

The truth of the formula should be obvious: if `p` holds for `3` then of course there is an `x` that makes `p` true, namely `3`! The rule of inference is symmetric with the one we just saw. If your conclusion is an existential, you can drop the quantifier and replace the bound variable by anything you want.

Basically, if you're trying to prove that something exists, just write down a term that computes it. For example, to prove

`(integerp x)` $\rightarrow$ $(\exists$ `y :  (+ x y)` $=$ `0)`

you could just drop the "$\exists$" and let `y` be `(- x)`, the negative of `x`. Then you're left to prove

`(integerp x)` $\rightarrow$ `((+ x (- x))` $=$ `0).`

The claim is that if you succeed in proving that, then you know

`(integerp x)` $\rightarrow$ $(\exists$ `y :  (+ x y)` $=$ `0)`

because the `y` that is alleged to exist has been exemplified.

• **Question 278**: The foregoing discussion of how to handle an "$\exists$" in the conclusion doesn't quite show the symmetry we talked about. When we instantiate a "$\forall$" in the hypothesis,

we get to copy the hypothesis and add an instance as a new conjunct, e.g., we transform the hypothesis `(∀ x: (p x))` to `((p 3) ∧ (∀ x: (p x)))`. What must we do for an "∃" in the conclusion? •

• **Question 279**: Give an example theorem whose proof naturally exploits the symmetric property of "∃" described in the previous question. •

When you instantiate a universal hypothesis or existential conclusion you may have to be creative!

• **Question 280**: Explain the strategy for proving

```
(¬(endp x)) → (∃ e :  (mem e x))?
```

Hint: Since we haven't seen the rule of inference for quantifiers, you can't give a precise proof yet. But I've told you (above) how one goes about proving $\alpha \rightarrow$ (∃ v : β), namely, show me a substitution $\sigma = \{v \leftarrow \delta\}$ such that you can prove $\alpha \rightarrow \beta/\sigma$. •

Sometimes the "creativity" requires you to define a new function just to say what you're thinking.

• **QUESTION 281**: Explain the strategy for proving

```
(mem e x) → (∃ i :  (nth i x) = e)
```

In particular, what formula would you transform this into? **Hint**: See the hint from Question 280 but you'll need to define a new function to express your strategy! •

• **QUESTION 282**: Suppose `natsp` is an arity 1 function that recognizes lists of natural numbers, suppose `ordp` is an arity 1 function that recognizes lists in weakly ascending order such as `(1 2 2 4 12)`, and suppose `perm` is an arity 2 function that recognizes when two lists are permutations of each other as are `(1 2 2 4 12)` and `(4 1 2 12 2)`. How would you prove

```
(natsp x) → (∃ y :  (ordp y) ∧ (perm x y))?
```

Hint: See the hint from Question 281. •

Now let's go around to the other side. Suppose we have a "∀" in the conclusion. What do we do? How could we prove a formula that looks like this?

γ:
```
(p x) → (∀ y :  (q x y))
```

Another way to put the question is this: what other formula could we prove that would convince a skeptical friend that the formula above is always true? One answer is to prove that when `(p x)` is true then `(q x z)` is true, for some "complete unknown" `z`, a variable we know absolutely nothing about. That is, we could prove

γ':

```
(p x) → (q x z).
```

Suppose we proved γ'. Then why is γ true? Let's evaluate γ for some particular values of the universally quantified `y`. Is `(p x) → (q x 3)` true? Yes, because γ' is true. Is `(p x) → (q x 7)` true? Yes. Is `(p x) → (q x '(a b c))` true? Yes. Is `(p x) → (q x nil)` true? Yes. The reason is always the same: γ' is true. Put another way, if we *prove* γ', then it is true *for every* value in $\mathcal{D}$. But that is what γ says.

We didn't really have to introduce `z`. One way to prove

γ:
```
(p x) → (∀ y :  (q x y))
```

is to just drop the "∀" and prove

```
(p x) → (q x y)
```

instead. But this works only because *we don't know anything about* `y`. That is, the only place `y` appears in γ is in the scope of the "∀." Put another way, the variable `y` is not free in γ. Think about it!

If we wanted to prove this γ instead:

γ:
```
((p x) ∧ (p y)) → (∀ y :  (q x y)).
```

We could *not* just drop the "∀" and re-use "y" as our "complete unknown" because we *know something about* `y` in this γ. But we could transform γ with an alpha conversion to

γ':
```
((p x) ∧ (p y)) → (∀ z :  (q x z))
```

and then drop the "∀" to get this goal to prove

γ'':
```
((p x) ∧ (p y)) → (q x z).
```

Warning: A really common mistake is for students to think that

γ:
```
(p x) → (∀ y :  (q x y))
```

and

γ':
```
(p x) → (q x y)
```

are equivalent, i.e.,

```
 ((p x) → (∀ y :  (q x y)))
↔
 ((p x) → (q x y))
```

The above equivalence *does not hold!* What we're saying is that if you *prove* γ' then you can consider γ proved. Another way to put it is: to prove γ, you can try to prove γ'. Still another version of the basic idea is that if γ' is valid (*always true*) then γ is valid (always true).

But γ and γ' are not equivalent! In particular, it is not the case that $\gamma \leftrightarrow \gamma'$.

• **QUESTION 283**: Define `p` and `q` so that, for some particular `x` and `y`, γ' is true but γ is false. Note that if you can do this, then the "↔" doesn't hold. You may assume $\mathcal{D}$ is the set of all natural numbers. Exhibit definitions of `p` and `q`, and values for `x` and `y`. •

To restate the warning: don't just drop a universal quantifier in the conclusion and think you haven't changed the value of the formula!

The symmetric case, of an "∃" in the hypothesis, is handled analogously. So how would you convince a skeptical friend that

```
(∃ y :  (p x y)) → (q x)
```

is always true? You'd prove

```
(p x z) → (q x)
```

where `z` is a complete unknown. That is if the hypothesis tells you there is some `y` that makes `(p x y)` true, then you may assume `(p x z)` for some `z` you know nothing else about.

What do we do when the formula we're proving has nested quantifiers? Be careful! Order matters!

Suppose $\mathcal{D}$ consists of the integers. Then the following formula is false.

```
γ:
(∀ x :  (∃ y :  (+ x y) = 0)) → (1=2)
```

The conclusion of γ is manifestly false. So the only way γ could always be true is if the hypothesis is always false. But the hypothesis is true. The hypothesis says that for every integer `x` there is an integer `y` such that their sum is `0`. This is clearly true: let `y` be the negative of `x`. Read it again: for every integer `x`, there is an integer, namely `(- x)`, such that `(+ x (- x))` is `0`. So γ is not a theorem.

Now imagine trying to prove this non-theorem by dropping the existential quantifier. This transformation is not allowed! Our rule will insist that the "∃" we drop in the hypothesis be the *outermost* quantifier of that hypothesis. But let's persist and mis-use the rule to see what happens. So, after checking that `y` is a complete unknown, we just drop the "∃" to transform γ to

γ':
`(∀ x : (+ x y) = 0)` $\rightarrow$ `(1=2)`.

This formula is always true because the hypothesis is always false. In particular, if `(+ x y)` is `0` for all `x`, then `(+ -1 y)` is `0` and so is `(+ -2 y)`. But that means `y` is `1` and `y` is also `2`.

In a good logic, we won't be able to prove γ and we ought to be able to prove γ'. That means we must not be able to transform γ to γ' to prove γ!

The mistake we made in producing γ' is that we dropped an interior quantifier. All of the transformations we've sketched operate only on the outermost quantifier of a hypothesis or conclusion.

Warning: Don't mess with interior quantifiers!

Before we introduced quantifiers, our rules of inference all preserved propositional equivalence: if an old rule reduces ψ to ψ', then $\psi \leftrightarrow \psi'$ is provable. Thus, many our proofs took the form $\psi \leftrightarrow \psi_1 \leftrightarrow \ldots \leftrightarrow True$.

But with some of the new quantifier-equipped rules, this propositional equivalence no longer hold. For example, as I've already said (but am happy to repeat because it is so important): a legal way to prove `(∀ x : (p x))` will be to prove `(p z)`, where `z` is a "complete unknown." But those two formulas are not propositionally equivalent. The first certainly implies the second. But the second doesn't imply the first: it is not generally a theorem that `(p x)` $\rightarrow$ `(∀ x : (p x))`. To be very precise, the *truth* of `(p x)` under an assignment does not necessarily imply the truth of `(∀ x : (p x))` under that assignment; the *validity* of `(p x)` implies the validity of `(∀ x : (p x))`.

We need a way to denote, in our proofs, that ψ can be proved by proving ψ'. The notation we use is "$\psi \dashv \psi'$". In this notation, here is a legal proof sketch for ψ: $\psi \leftrightarrow \psi_1 \leftrightarrow \psi_2 \dashv \psi_3 \leftrightarrow \psi_4 \dashv True$. Starting at the right-hand end of the chain, it means from $True$ we can prove ψ_4, which is equivalent to ψ_3 from which we can prove ψ_2 which is equivalent to ψ_1 which is equivalent to our goal ψ.

Note: In usual treatments of logic, "$\psi' \vdash \psi$" is used to mean that ψ can be proved from ψ'. More generally and precisely, "$\psi_1, \ldots, \psi_k \vdash \psi$" means that ψ can be proved from the conjunction of the universal closures of the ψ_i. The symbol "$\vdash$" is called a *turnstyle*. By reversing it, "$\psi \dashv \psi'$" we allow ψ to be written on the left, and ψ' on the right. That allows proofs to be written as a sequence of formulas starting with the goal and ending with $True$, each of which is sufficient to prove its neighbor on the left: $\psi \leftrightarrow \psi' \leftrightarrow \psi'' \dashv \psi''' \leftrightarrow \ldots True$.

In summary, to change our old logic – where there were no quantifiers – to a logic where there are quantifiers, we have to

- change the definition of "instantiation" so that when we instantiate a formula we don't instantiate or capture free variables (remember: you can't instantiate bound variables and you can't let your free variables be captured)
- change our old rules for replacement and rewriting so we don't use them inside the scope of a quantifier (remember: you can't substitute equals for equals inside of quantifiers without checking extra conditions)

- introduce the alpha-conversion rule
- introduce rules for handling a "∀" in the hypothesis and a "∃" in the conclusion, and
- introduce rules for handling an "∃" in the hypothesis and an "∀" in the conclusion.

Only the two rules that deal with ∀ in the conclusion and ∃ in the hypothesis require the "⊣" notation. All the others preserve propositional equivalence.

7.6 Rules of Inference for Quantified Formulas (Assignment 16: 7 days)

Henceforth, when we use the notation ϕ/σ or speak of *instantiating* formula ϕ with substitution σ, we mean as substitution is defined on page 201. Please re-read the statements of the Tautology rule (page 128), the Cases rule (page 129), the Constant rule (page 130), the Computation rule (page 130), and the Induction rule (page 170) with this new definition of instantiation in mind. These rules do not change otherwise.

We will re-state the Rewrite and Hypotheses rules below to deal with the possible presence of quantifiers.

Rule of Inference Rewrite (for Quantified Formulas) ("qRe"):

Summary: This rule allows you to rewrite part of the goal ψ using a previously established theorem (including an axiom or definition) ϕ.

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c so that the term you wish to replace is in goal's conclusion ψ_c
2. let π be the address in ψ_c pointing to the subterm you want to replace
3. factor the previously proved theorem ϕ into a hypothesis ϕ_h and conclusion ϕ_c
4. factor ϕ_c into pattern α, replacement β, and maintained equivalvence *eqv* such that the term you want to replace (at π in ψ_c) is an instance of α under some substitution σ
5. if *eqv* is `iff`, confirm that address π admits a propositional replacement in ψ_c
6. if π is in the scope of a quantifier, prove (ϕ_h/σ); otherwise, prove $\psi_h \rightarrow (\phi_h/\sigma)$
7. obtain ψ'_c by replacing the term at π in ψ_c by β/σ

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow \psi'_c$.

End of Rule

Note: The Rewrite rule for quantified formulas is just like its quantifier-free version except for step 6. In the quantified version of the rule, if you are replacing a target within the scope of a quantifier, you must relieve the hypotheses of the theorem without using any of the hypotheses of the goal.

Rule of Inference Hypothesis (for Quantified Formulas) (or "qHyp"):

Summary: This rule allows you to use one of the hypotheses of the goal ψ by replacing some term in its conclusion with an equivalent term.

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c
2. let δ be one of the conjuncts of ψ_h (a hypothesis of goal ψ)
3. factor hypothesis δ into pattern α, replacement β, and maintained equivalence *eqv*
4. let π be an address in ψ_c at which you find α
5. if *eqv* is `iff`, confirm that address π admits a propositional replacement in ψ_c
6. confirm that no variable in the bound variable list at π is free in δ
7. obtain ψ_c' by replacing the term at π in ψ_c by β

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow \psi_c'$.

Note: The Hypothesis rule for quantified formulas has an extra step (namely 6) that prevents you from using hypotheses whose variables are captured.

We add five new rules of inference to deal with quantifiers.

Rule of Inference Alpha Conversion ("α-Conv")

Summary: This rule allows you to change the name of the bound variable of a quantified expression occuring in a goal ψ.

Description: If you can carry out all of the following steps:

1. let π be a legal address in ψ pointing to a quantified expression $(\mathcal{Q}\ v\ :\ \phi)$, where $\mathcal{Q}$ is either "$\forall$" or "$\exists$"
2. let u be a variable symbol that is not free in ϕ
3. let ϕ' be the result of replacing all free occurrences of v in ϕ by u
4. let ψ' be the result of replacing the expression at π in ψ by $(\mathcal{Q}\ u\ :\ \phi')$

then $\psi \leftrightarrow \psi'$.

End of Rule

Rule of Inference Universal Hypothesis ("$\forall$-Hyp)

Summary: This rule allows you to copy and instantiate the outermost universal quantifier in the jth hypothesis of your goal ψ, converting ψ from $(\psi_{h_1} \land \ldots \land (\forall v : \phi) \land \ldots \land \psi_{h_k}) \rightarrow \psi_c$ to $(\psi_{h_1} \land \ldots \land \phi/\sigma \land \phi_{h_j} \land \ldots \land \psi_{h_k}) \rightarrow \psi_c$.

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c
2. let the conjuncts of ψ_h be $\psi_{h_1}, \ldots, \psi_{h_j}, \ldots, \psi_{h_k}$
3. choose j so that ψ_{h_j} is of the form $(\forall\ v\ :\ \phi)$
4. choose any term α and let σ be the substitution $\{v \leftarrow \alpha\}$
5. let ψ'_h be the result of replacing the jth conjunct of ϕ_h by $(\phi/\sigma \land \phi_{h_j})$

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi'_h \rightarrow \psi_c$.

End of Rule

Rule of Inference Existential Conclusion ("$\exists$-Concl)

Summary: This rule allows you to copy and instantiate the existentially quantified conclusion in a goal ψ, converting $(\psi_h \rightarrow (\exists v : \phi))$ to $(\psi_h \rightarrow (\phi/\sigma \lor (\exists v : \phi)))$.

Description:

If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c so that ψ_c is of the form $(\exists\ v\ :\ \phi)$
2. choose any term α and let σ be the substitution $\{v \leftarrow \alpha\}$

then $\psi \leftrightarrow \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow (\phi/\sigma \lor \psi_c)$.

End of Rule

Rule of Inference Universal Conclusion ("$\forall$-Concl")

Summary: This rule allows you to drop the outermost universal quantifier in the conclusion of your goal, converting ψ from $\psi_h \rightarrow (\forall v : \phi)$ to $\psi_h \rightarrow \phi$ provided v is not free in ψ. *This rule does not preserve propositional equivlence, just "$\dashv$."*

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c so that ψ_c is of the form $(\forall\ v\ :\ \phi)$

2. confirm that v is not free in ψ (if v is free in ψ then you could use α-Conv first to rename v to some other variable symbol)

then $\psi \dashv \psi'$, where ψ' is any refactoring of $\psi_h \rightarrow \phi$.

End of Rule

Rule of Inference Existential Hypothesis ("∃-Hyp")

Summary: This rule allows you to drop the outermost existential quantifier in the jth hypothesis of your goal ψ, converting ψ from $(\psi_{h_1} \wedge \ldots \wedge (\exists v : \phi) \wedge \ldots \wedge \psi_{h_k}) \rightarrow \psi_c$ to $(\psi_{h_1} \wedge \ldots \wedge \phi \wedge \ldots \wedge \psi_{h_k}) \rightarrow \psi_c$, provided v is not free in ψ. *This rule does not preserve propositional equivlence, just "⊣."*

Description: If you can carry out all of the following steps:

1. factor the goal ψ into a hypothesis ψ_h and a conclusion ψ_c
2. let the conjuncts of ψ_h be $\psi_{h_1}, \ldots, \psi_{h_j}, \ldots, \psi_{h_k}$
3. choose j so that ψ_{h_j} is of the form `(∃ v : φ)`
4. confirm that v is not free in ψ (if v is free in ψ then you could use α-Conv first to rename v to some other variable symbol)
5. let ψ'_h be the result of replacing the jth conjunct of ϕ_h by ϕ

then $\psi \dashv \psi'$, where ψ' is any refactoring of $\psi'_h \rightarrow \phi_c$.

End of Rule

7.7 Practice Proofs (Assignment 16 cont'd)

In the problems below, the function symbols `p` and `q` are undefined and used with different arities in different problems. The other functions mentioned in these problems are defined at the end of this section.

This section contains much more material than just problems. It teaches you a lot about quantifier manipulation. I strongly recommend that you read every problem and the text in between without thinking about your homework assignment! I show you how these theorems about undefined `p`s and `q`s can be used to do quantifier manipulation about terms we really care about.

I'll do two problems as a reminder of what I'm looking for in the way of quantifier proofs.

• **Question 284**:

```
(∀ x : (∃ y : (p x) ∧ (q y))) → (∀ x : (p x))
```

My Proof:

`(∀ x : (∃ y : (p x) ∧ (q y))) → (∀ x : (p x))`

⊣ {∀-Concl (see note 1 below)}

`(∀ x : (∃ y : (p x) ∧ (q y))) → (p a)`

↔ {∀-Hyp (see note 2 below)}

`((∃ y : (p a) ∧ (q y)) ∧ (∀ x : (∃ y : (p x) ∧ (q y)))) → (p a)`

⊣ {∃-Hyp (see note 3 below)}

`((p a) ∧ (q b) ∧ (∀ x : (∃ y : (p x) ∧ (q y)))) → (p a)`

↔ {Taut (see note 4 below)}

True

□

Notes: In these notes I explain what motivated me to take each step. I don't expect you to provide such notes with your proofs. I recommend reading a formula from the proof and the justification, then comparing the upper formula to the next one to see what transformation I made, and then reading the note for that justification.

1. I'm trying to prove a universal conclusion. I can prove it by proving it for a complete unknown, `a`. I tend to use variables from the beginning of the alphabet for those that have to be new.

2. I have a universal hypothesis that can tell me about `(p a)`. So I make that explicit by using the ∀-Hyp rule and choosing `a` for `x`. I couldn't have used ∀-Hyp effectively had I tried to before I introduced `a`! I *had* to take step 1 and then step 2 in that order. Step 1 introduced a complete unknown and then step 2 provided a useful fact about that unknown. I didn't have a name for the unknown before step 1. If you try to do the steps in the other order and "anticipate" the introduction of the name "`a`" you will be foiled by the ∀-Concl rule's insistence on a complete unknown. With the name "`a`" no longer unknown, it would have introduced a different variable.

3. I have an existential hypothesis. It is actually irrelevant. But my `(p a)` is buried inside the ∃-term. So I use ∃-hyp to get rid of the quantifier. I have to choose a new variable for `y` and I choose `b`. I could have chosen `y` for `y`, because `y` is not free in the formula, but I tend to choose new names that are not involved at all, so that I do not have to pay attention to the scopes of quantifiers after choosing the name.

4. Now my formula is a tautology of the form `(u ∧ w) → u`. So I'm done.

As you can see, the order in which you apply the quantifier elimination rules is important.

Here is another version of the same proof. Recall that the ∀-Hyp rule copies the universally quantified hypothesis and drops the quantifier from one of the copies. It preserves the original quantified hypothesis in case you need another instance. But I know I don't actually

need it again. So I drop it in the proof below. I can do that because of the Tautology `(u →  v) → ((u ∧ w) → v)`. The formula I write after the ∀-Hyp justification below is no longer propositionally equivalent to the earlier one, but implies it. Note I changed the connector on this line from "↔" to "←" to denote this.

If you need to type "←" in ASCII, use "<--".

Theorem:
`(∀ x : (∃ y : (p x) ∧ (q y))) → (∀ x : (p x))`

My Proof:

`(∀ x : (∃ y : (p x) ∧ (q y))) → (∀ x : (p x))`
⊣ {∀-Concl}
`(∀ x : (∃ y : (p x) ∧ (q y))) → (p a)`
← {∀-Hyp and Taut}
`(∃ y : (p a) ∧ (q y)) → (p a)`
⊣ {∃-Hyp}
`((p a) ∧ (q b)) → (p a)`
↔ {Taut}
True
□

Here is a rather long but illustrative *inductive* proof about a quantified expression. I will prove the hard part of the theorem:

Theorem:
`(perm x y)` ↔ (∀e : `(how-many e x)` = `(how-many e y)`) •

The relevant definitions are at the end of this section. I'll consider the two directions of the "↔" separately.

The left-to-right implication,

Theorem γ_1**:**
`(perm x y) → (∀ e : (how-many e x) = (how-many e y)),`

is the easier. In particular, by ∀-Concl we could convert this to

γ_1':
`(perm x y) → (how-many e x) = (how-many e y)`

which is a formula we could in principle have proved before we introduced quantifiers.

It should also be clear that the converse of γ_1'

γ_2':
`(how-many e x) = (how-many e y) → (perm x y)`

is *not valid*. For example, if `e` is `1` and `x` is the list `(1 2 3)` and `y` is the list `(1 2 2)`, then

γ_2' is false. However, it is the case that

γ_2:
```
(∀ e :  (how-many e x) = (how-many e y)) → (perm x y)
```

is a theorem. Clearly, if we could prove the ACL2 formula γ_1' and the quantified formula γ_2 then we would know:

```
(perm x y) ↔ (∀ e :  (how-many e x) = (how-many e y)).
```

This is a wonderful result because it allows us to reason about the rather complicated function `perm` by reasoning about counting. Put another way, it allows us to look at `perm` in two ways. The first is the recursive definition of `perm`, which is a *program* we can execute. It considers a finite number of elements and announces whether two lists are permutations of each other. The program `perm` involves membership and deletion of elements from lists. The second way of looking at `perm` is the universally quantified `how-many` formula. `How-many` is very simple: it just counts how many times an element appears in a list. But the quantified expression can't be executed because it considers an infinite number of possible elements to count. Perhaps surprisingly, it is easier to reason about the quantified `how-many` expression. But you can't use it in a program.

I assume you can prove γ_1' and so I'll focus on γ_2.

For my proof of γ_2 I will use the following quantifier-free lemmas, all of which you should be able to prove.

`how-many-endp:` {Trivial case from def of `how-many`}
```
(endp x) → ((how-many e x) = 0)
```

`not-endp-how-many:` {Trivial case from def of `how-many`}
```
  (¬ (endp x))
→
  ((how-many e x)
   =
   (if (equal e (first x))
       (1 + (how-many e (rest x)))
       (how-many e (rest x))))
```

`not-endp-perm:` {Trivial case from def of `perm`}
```
  (¬(endp x))
→
  ((perm x y)
   ↔
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y))))
```

`how-many-first-0:` {Trivial from def of `how-many`}
```
(0 = (how-many (first y) y)) → (endp y)
```

`natp-how-many:` {Nice little induction. Try it!}
`(natp (how-many e x))`

`non-0-how-many:` {Nice result linking `how-many` and `mem`}
`(0 < (how-many e x))` $\leftrightarrow$ `(mem e x)`

`how-many-general:` {Very cool generalization of `how-many`. Try it!}

```
(how-many e x)
=
(if (mem e x)
    (+ 1 (how-many e (del1 e x)))
    0)))
```

`how-many-del1:` {Nice little fact. Try it!}

```
  (e ≠ d)
→
  ((how-many e (del1 d y))
   =
   (how-many e y))
```

None of these involve quantification and the ones noted are worth your attention. But let me get on with my proof.

Theorem `perm-how-many:`
`(`$\forall$ `e : (how-many e x) = (how-many e y))` $\rightarrow$ `(perm x y).`

Proof
`Induct on x, using` $\sigma_1 = \{\texttt{x} \leftarrow \texttt{(rest x)}, \texttt{y} \leftarrow \texttt{(del1 (first x) y)}\}$.

Base Case:

```
   (endp x)
 →
   ((∀ e : (how-many e x) = (how-many e y))
    →
    (perm x y))
```

$\leftrightarrow$ {Promotion}

```
  ((endp x)
   ∧
   (∀ e : (how-many e x) = (how-many e y)))
 →
   (perm x y)
```

$\leftrightarrow$ { `perm-endp`}

```
  ((endp x)
   ∧
   (∀ e : (how-many e x) = (how-many e y)))
```

```
 →
   (endp y)
```

{*Strategy: I need to prove* `(endp y)`*. I know* `x` *is empty. The quantified hypothesis tells me that, regardless of what* `e` *I choose,* `e` *occurs as many times in* `x` *as in* `y`*. I must choose something for* `e` *to demonstrate that* `y` *is empty. It must be something that must be in* `y` *if* `y` *is non-empty.*}

⊣ {∀-Hyp (choose `(first y)` for `e`}

```
  ((endp x)
   ∧
   ((how-many (first y) x) = (how-many (first y) y)))
 →
   (endp y)
```

↔ { `how-many-endp`}

```
  ((endp x)
   ∧
   (0 = (how-many (first y) y)))
 →
   (endp y)
```

↔ {`how-many-first-0`}

True

{*Strategy: That finishes the Base Case. Before proving the Induction Step, consider the* σ_1 *I chose:* {`x` ← `(rest x)`, `y` ← `(del1 (first x) y)`}*. It "unwinds"* `(perm x y)`*. In particular, note that the call of* `perm` *in my induction hypothesis below is exactly the recursive call of* `perm` *I'll get when I expand* `(perm x y)`*.*

The induction step to prove γ_2 *is, of course, just* `((¬(endp x) ∧` γ_2/σ_1`) →` γ_2`)`*. But since* γ_2 *has a quantifier in it, the Induction Hypothesis will have a quantifier in it, making it look more complicated than usual. And I must pay attention when applying* σ_1 *to the quantified* γ_2*, athough in this case, no captures are threatened.*

Since γ_2 *is of the form* `((∀ e : ...) → ...)`*, the Induction Step is of the form:*

`((¬(endp x) ∧ ((∀ e :  ...) → ...)) →` γ_2`)`

and not of the form

`((¬(endp x) ∧  (∀ e :  ...)     ... ) →` γ_2`)`

Pay attention to the parentheses. The quantifier in the hypothesis of the goal above is "buried" in an implication. So I can't just do ∀-Hyp on it! I have to "lift" that quantifier out somehow. You'll see that when I do, it becomes an existential! But I'm getting ahead of myself.

In any case, my first few moves are conventional. I just promote and open the conclusion to see what I've got to prove. Then I'll think about the quantifiers.

Induction Step:

```
   ((¬(endp x))
    ∧
    ((∀ e :
       ((how-many e (rest x))
         =
        (how-many e (del1 (first x) y))))
     →
     (perm (rest x) (del1 (first x) y))))
 →
   ((∀ e :  (how-many e x) = (how-many e y)) → (perm x y))
```

↔ {Promotion}

```
   ((¬(endp x))
    ∧
    ((∀ e :
       ((how-many e (rest x))
         =
        (how-many e (del1 (first x) y))))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))
 →
   (perm x y)
```

↔ {not-endp-perm}

```
   ((¬(endp x))                                           {Hyp 1}
    ∧
    ((∀ e :                                  {Hyp 2 (IH):  IH_h → IH_c}
       ((how-many e (rest x))
         =
        (how-many e (del1 (first x) y))))
     →
     (perm (rest x) (del1 (first x) y)))                  {IH_c}
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))             {Hyp 3}
 →
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y)))
```

{*Strategy: Look at the conclusion. We need to prove a* `mem`*-expression and a* `perm`*-expression. We have three hyps, Hyp 1, Hyp 2, and Hyp 3. Hyp 2 is our Induction Hypothesis and is of the form ($IH_h \rightarrow IH_c$). Note that IH_c is the very* `perm`*-expression we wish to prove. That*

is due to my choice of σ_1.

So there will be two big phases of the proof. (a) We can use Hyp 3 to establish the `mem`*-expression, because we know* `(first x)` *appears non-0 times in the non-empty* `x` *and so must appear non-0 times in* `y`*. (b) To use* ($IH_h \rightarrow IH_c$) *to prove the* `perm`*-expresion we have to detatch* IH_c *from IH with Forward Chaining from Hyp 3. But* IH_h *and Hyp 3 are different. We'll need to rearrange those two appropriately.*

Resuming the proof from where we left off, and entering phase (a), ...}

↔ {∀-Hyp, choosing `(first x)` for `e`}

```
   ((¬(endp x))
    ∧
    ((∀ e :
       ((how-many e (rest x))
        =
        (how-many e (del1 (first x) y))))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    ((how-many (first x) x) = (how-many (first x) y))
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))
 →
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y)))
```

↔ {`not-endp-how-many`, Equal Reflexivity, `if-t`}

```
   ((¬(endp x))
    ∧
    ((∀ e :
       ((how-many e (rest x))
        =
        (how-many e (del1 (first x) y))))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    ((1 + (how-many (first x) (rest x))) = (how-many (first x) y))
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))
 →
   ((mem (first x) y)
    ∧
    (perm (rest x) (del1 (first x) y)))
```

↔ {Arithmetic and `natp-how-many`}

```
  ((¬(endp x))
   ∧
   ((∀ e :
      ((how-many e (rest x))
        =
       (how-many e (del1 (first x) y))))
     →
    (perm (rest x) (del1 (first x) y)))
   ∧
   (0 < (how-many (first x) y))
   ∧
   (∀ e :  (how-many e x) = (how-many e y)))
 →
  ((mem (first x) y)
   ∧
   (perm (rest x) (del1 (first x) y)))
```

↔ {`non-0-how-many`}

```
  ((¬(endp x))
   ∧
   ((∀ e :
      ((how-many e (rest x))
        =
       (how-many e (del1 (first x) y))))
     →
    (perm (rest x) (del1 (first x) y)))
   ∧
   (mem (first x) y)
   ∧
   (∀ e :  (how-many e x) = (how-many e y)))
 →
  ((mem (first x) y)
   ∧
   (perm (rest x) (del1 (first x) y)))
```

↔ {Hyp and Short-Circuit}

```
  ((¬(endp x))
   ∧
   ((∀ e :
      ((how-many e (rest x))
        =
       (how-many e (del1 (first x) y))))
```

```
      →
      (perm (rest x) (del1 (first x) y)))
     ∧
     (mem (first x) y)
     ∧
     (∀ e :  (how-many e x) = (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
```

{*Strategy: That completes phase (a). Now we work on the* `perm`*-expression as phase (b). Note that if we could forward chain into the second hypothesis, we would be done because it's conclusion is the* `perm`*-expression we wish to establish. To forward chain, we need to make the third hypothesis above match the hypothesis of the second hypothesis. We do that by rearranging both. But we can't work freely within the scope of the quantifier in the second hypothesis. Furthermore, that quantifier is in the hypothesis of a hypothesis so we can't use* ∀*-Hyp on it! We have to lift that quantifier out so that it at the level of our hypotheses. Then we can eliminate it, and then we can work on it. Resuming the proof from where we left off, and entering phase (b), ...*}

```
↔                                                        {implies-all}
   ((¬(endp x))
    ∧
    (∃ e :
      ((how-many e (rest x))
        =
       (how-many e (del1 (first x) y)))
      →
      (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
↔                        {∃-Hyp (note e is a “complete unknown”) }
   ((¬(endp x))
    ∧
    (((how-many e (rest x))
      =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    (∀ e :  (how-many e x) = (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
⊣                                       {∀-Hyp (choose e for e)}
   ((¬(endp x))
    ∧
    (((how-many e (rest x))                                  {IH_h}
      =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e x) = (how-many e y)))                      {Hyp 3}
 →
```

```
   (perm (rest x) (del1 (first x) y))
```

{*Strategy: Now I've gotten rid of the quantifiers and can go to work on trying to make* IH_h *and Hyp 3 match. That means I need to relate* `(how-many e x)` *to* `(how-many e (rest x))` *and I need to relate* `(how-many e y)` *to* `(how-many e (del1 (first x) y))`*. The first is easy but the relationship depends on whether* `e` *is* `(first x)`*. The second is less obvious, but depends on the same question. If* `e` *is* `(first x)`*, then* `(del1 (first x) y)` *has one less* `e` *than* `y`*, and if* `e` *is not* `(first x)`*, the number of* `e`*s in both is the same. So I need to know whether* `e` *is* `(first x)`*. That suggests I do a case split and consider the two cases separately. When you see me finally do the Forward Chaining steps in the two cases below you'll know that I've succeeded in making* IH_h *and Hyp 3 match! ...*}

$\leftrightarrow$ {Cases on whether `(e=(first x))` is True or False}

Case 1:

```
   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many e (rest x))
       =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e x) = (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
```

$\leftrightarrow$ {Hyp (4 times)}

```
   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
       =
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many (first x) x) = (how-many (first x) y)))
 →
```

```
   (perm (rest x) (del1 (first x) y))

↔                                         {not-endp-how-many}

   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
      =
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((if (equal (first x) (first x))
         (1 + (how-many (first x) (rest x)))
         (how-many (first x) (rest x)))
     =
     (how-many (first x) y)))
 →
   (perm (rest x) (del1 (first x) y))

↔                                         {Equal Reflexive, if-t}

   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
      =
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((1 + (how-many (first x) (rest x)))
     =
     (how-many (first x) y)))
 →
   (perm (rest x) (del1 (first x) y))

↔                                         {how-many-general}

   ((e = (first x))
```

```
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
      =
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((1 + (how-many (first x) (rest x)))
     =
     (if (mem (first x) y)
         (1 + (how-many (first x) (del1 (first x) y)))
         (how-many (first x) (del1 (first x) y)))))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Hyp and `if-t`}

```
   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
      =
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((1 + (how-many (first x) (rest x)))
     =
     (1 + (how-many (first x) (del1 (first x) y)))))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Arithmetic and `natp-how-many`}

```
   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many (first x) (rest x))
      =
```

```
      (how-many (first x) (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many (first x) (rest x))
     =
     (how-many (first x) (del1 (first x) y))))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Forward Chaining (!!!)}

```
   ((e = (first x))
    ∧
    (¬(endp x))
    ∧
    (perm (rest x) (del1 (first x) y))
    ∧
    (mem (first x) y)
    ∧
    ((how-many (first x) (rest x))
      =
      (how-many (first x) (del1 (first x) y))))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Taut}

True

Case 2:

```
   ((e ≠ (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many e (rest x))
      =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e x) = (how-many e y)))
```

```
 →
   (perm (rest x) (del1 (first x) y))

↔                                              {not-endp-how-many}

   ((e ≠ (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many e (rest x))
      =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((if (equal e (first x))
         (1 + (how-many e (rest x)))
         (how-many e (rest x)))
     =
     (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))

↔                                              {Hyp and if-nil}

   ((e ≠ (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many e (rest x))
      =
      (how-many e (del1 (first x) y)))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e (rest x))
     =
     (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))

↔                                              {how-many-del1}
```

```
   ((e ≠ (first x))
    ∧
    (¬(endp x))
    ∧
    (((how-many e (rest x))
      =
      (how-many e y))
     →
     (perm (rest x) (del1 (first x) y)))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e (rest x))
     =
     (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Forward Chaining (!!!)}

```
   ((e ≠ (first x))
    ∧
    (¬(endp x))
    ∧
    (perm (rest x) (del1 (first x) y))
    ∧
    (mem (first x) y)
    ∧
    ((how-many e (rest x))
     =
     (how-many e y)))
 →
   (perm (rest x) (del1 (first x) y))
```

↔ {Taut}

True

□

Prove or show counterexamples and/or countermodels to the alleged theorems below.

• **Question 285**:

`(∃ v :  (p v)) ↔ (¬(∀ v :  (¬(p v)))).`

• **Question 286**:

(p x) $\rightarrow$ ($\forall$ x : (p x)) •

• **Question 287**:

($\forall$ x : (p x)) $\rightarrow$ (p x) •

• **Question 288**:

(p x) $\rightarrow$ ($\exists$ x : (p x)) •

• **QUESTION 289**:

($\exists$ x : (p x)) $\rightarrow$ (p x) •

• **Question 290**:

($\forall$ x : (p x)) $\rightarrow$ ((p i) $\wedge$ (p (cons u v))) •

• **Question 291**:

((p (+ 1 x)) $\vee$ (p (rest v))) $\rightarrow$ ($\exists$ x : (p x)) •

The following problems show how quantifiers can – and cannot – be "moved" around conjunctions and disjunctions. Study these carefully even if they are not assigned on the homeworks because they suggest transformations that are useful in other proofs.

• **QUESTION 292**: (`all-and-1`)

($\forall$ x : ((p x y) $\wedge$ (q x y))) $\rightarrow$ (($\forall$ x : (p x y)) $\wedge$ ($\forall$ x : (q x y))) •

• **QUESTION 293**: (`all-and-2`)

(($\forall$ x : (p x y)) $\wedge$ ($\forall$ x : (q x y))) $\rightarrow$ ($\forall$ x : ((p x y) $\wedge$ (q x y))) •

Note that the combination Questions 292 (page 231) and 293 (page 231) show that

```
all-and:
  ((∀ x : (p x y)) ∧ (∀ x : (q x y)))
↔
  (∀ x : ((p x y) ∧ (q x y))).
```

Thus, the universal quantifier distributes over conjunction.

This can be used to replace one universally quantified expression by another. The `all-and` theorem literally requires the two conjuncts to be "`(p x y)`" and "`(q x y)`" and thus looks fairly useless. But the fact that it can be proved without knowing anything about `p` and `q` means that we could prove the analogous theorem about terms of real interest. For example, in exact analogy to the proofs in Questions 292 (page 231) and 293 (page 231) we could prove:

```
  ((∀ e : (mem e (app a b))) ∧ (∀ e : (natp e)))
```

```
↔
  (∀ e :  (mem e (app a b)) ∧ (natp e))
```

Here's a way to see that. Since we proved Questions 292 (page 231) and 293 (page 231) without any knowledge of `p` and `q`, they could have been defined to be:

```
(defun p (x y)
  (mem x (app (first y) (second y))))

(defun q (x y)
  (natp x))
```

Thus, from the "useless" theorem

```
  ((∀ x :  (p x y)) ∧ (∀ x :  (q x y)))
↔
  (∀ x :  ((p x y) ∧ (q x y)))
```

we could derive, by Alpha Conversion,

```
  ((∀ e :  (p e y)) ∧ (∀ e :  (q e y)))
↔
  (∀ e :  ((p e y) ∧ (q e y)))
```

and by instantiation

```
  ((∀ e :  (p e (list a b))) ∧ (∀ e :  (q e (list a b))))
↔
  (∀ e :  ((p e (list a b)) ∧ (q e (list a b))))
```

and then by expanding the definitions of `p` and `q`

```
  ((∀ e :  (mem e (app a b))) ∧ (∀ e :  (natp e)))
↔
  (∀ e :  (mem e (app a b)) ∧ (natp e)).
```

Thus, even though `all-and` appears practically useless (since it talks about `p` and `q` and we don't really care about them) it actually serves as kind of template or schema giving us a way to distribute universal quantifiers over conjunctions.

I will allow you to cite `all-and` and the related quantifier manipulation theorems below as justification for legitimate quantifier rearrangements. But make sure your rearrangements can be explained by an argument (which you don't have to write down) like that given above. Of particular concern is where the bound and free variables appear. The surest way to convince yourself is to make sure you could define the `p` and `q` in the cited theorem so that the theorem justifies the transformation made.

• **Question 294**: (`exist-and-1`)

$(\exists$ x : ((p x y) $\wedge$ (q x y))) $\rightarrow$ (($\exists$ x : (p x y)) $\wedge$ ($\exists$ x : (q x y))) •

• **Question 295**: You won't be able to prove this because it isn't valid! But trying to prove it will teach you about the rules of inference and I encourage you to follow your instincts and see where the proof breaks down. In any case, show a countermodel and counterexample.

(($\exists$ x : (p x y)) $\wedge$ ($\exists$ x : (q x y))) $\rightarrow$ ($\exists$ x : ((p x y) $\wedge$ (q x y))) •

Thus, the existential quantifier does *not* distribute over conjunctions, though you do get one of the two directions: if you know there is an `x` that satisfies `p` and `q`, then you know there is an `x` that satisfies `p` and there is an `x` that satisfies `q`. But you don't know the converse and if you answer Question 295 (page 233) you'll understand why.

• **QUESTION 296**: You won't be able to prove this because it isn't valid! But trying to prove it will teach you about the rules of inference and I encourage you to follow your instincts and see where the proof breaks down. In any case, show a countermodel and counterexample.

($\forall$ x : ((p x y) $\vee$ (q x y))) $\rightarrow$ (($\forall$ x : (p x y)) $\vee$ ($\forall$ x : (q x y))) •

and• **Question 297**: (`all-or-2`)

(($\forall$ x : (p x y)) $\vee$ ($\forall$ x : (q x y))) $\rightarrow$ ($\forall$ x : ((p x y) $\vee$ (q x y))) •

Thus, the universal quantifier does *not* distribute over disjunction, but you get one of the directions: if you know that all `x` satisfy `p` or that all `x` satisfy `q`, then clearly all `x` satisfy their disjunction. But not vice versa: if all `x` satisfy either `p` or `q`, it doesn't mean that all `x` satisfy just one of them. By thinking of an answer to Question 296 (page 233) you'll see why.

• **Question 298**: (`exists-or-1`)

($\exists$ x : ((p x y) $\vee$ (q x y))) $\rightarrow$ (($\exists$ x : (p x y)) $\vee$ ($\exists$ x : (q x y))) •

• **Question 299**: (`exists-or-2`)

(($\exists$ x : (p x y)) $\vee$ ($\exists$ x : (q x y))) $\rightarrow$ ($\exists$ x : ((p x y) $\vee$ (q x y))) •

Thus, the existential quantifier distributes over disjunction.

```
exists-or:
  ((∃ x :  (p x y)) ∨ (∃ x :  (q x y)))
↔
  (∃ x :  ((p x y) ∨ (q x y))).
```

The symmetries – universal distributes over conjuncts, existential distributes over disjuncts, and symmetric implications hold in the "mixed" cases – make intuitive sense if you think of universal quantification as a "big $\wedge$," existential as a "big $\vee$," and realize that conjunction implies disjunction but not vice versa.

• **Question 300**: (`all-not`, `exists-not`, `not-all`, and `not-exists`)

It should be clear that neither universal nor existential quantification distributes over negation! The following is *not* a theorem!

```
(∀ x : (¬ (p x y))) ↔ (¬ (∀ x : (p x y)))
```
{Wrong!}

In fact, you should remember that when you move a quantifier through a negation, the quantifier changes its "kind", from universal to existential and vice versa. Prove the first but remember them all:

```
all-not:
(∀ x : (¬ (p x y))) ↔ (¬ (∃ x : (p x y)))

exists-not:
(∃ x : (¬ (p x y))) ↔ (¬ (∀ x : (p x y)))

not-all:
(¬ (∀ x : (p x y))) ↔ (∃ x : (¬ (p x y)))

not-exists:
(¬ (∃ x : (p x y))) ↔ (∀ x : (¬ (p x y))) •)
```

• **Question 301**: (`implies-all`)

```
  ((∀ x : (p x y)) → (q y))
↔
  (∃ x : (p x y) → (q y))
```

Hint: Prove and then use appropriately the lemma that `(q y) ↔ (∃ x : (q y))`. •

Warning: The `implies-all` theorem above is subtle because `(q y)` is in the scope of the `(∃ x : ...)` but not in the scope of the `(∀ x : ...)`. Note that `(q y)` doesn't have an `x` in it. To be more precise, $((\forall \text{ x} : \psi) \rightarrow \phi)$ is equivalent to $(\exists \text{ x} : \psi \rightarrow \phi)$, *provided* `x` is not free in ϕ. The lemma given in the hint makes this clear too.

The next series of questions sheds light on when we can rearrange the order of a string of quantifiers. For example can we rearrange `(∀ x : (∀ y : (∃ z : ...)))` to `(∀ y : (∀ x : (∃ z : ...)))` or to `(∃ z : (∀ x : (∀ y : ...)))`? These are sometimes called "quantifier swap" theorems because they concern switching the order of two adjacent quantifiers.

• **Question 302**: (`all-all`)

```
(∀ y : (∀ x : (p x y z))) → (∀ x : (∀ y : (p x y z))) •
```

Of course, from the theorem in Question 302 (page 234), by Alpha Conversion, we know the converse:

```
(∀ x : (∀ y : (p x y z))) → (∀ y : (∀ x : (p x y z))) •
```

So we have the equivalence:

```
  (∀ x : (∀ y : (p x y z)))
```

```
↔
  (∀ y : (∀ x : (p x y z)))
```

and can swap two adjacent universal quantifiers.

• **Question 303**: (`exists-exists`)

`(∃ y : (∃ x : (p x y z))) → (∃ x : (∃ y : (p x y z)))` •

Following the same line of reasoning used after Question 302 (page 234), we see from Question 303 (page 235) that we can also swap adjacent existentials.

```
  (∃ y : (∃ x : (p x y z)))
↔
  (∃ x : (∃ y : (p x y z)))
```

• **Question 304**: (`exists-all-1`)

`(∃ y : (∀ x : (p x y z))) → (∀ x : (∃ y : (p x y z)))` •

• **Question 305**: You won't be able to prove this because it isn't valid! But trying to prove it will teach you about the rules of inference and I encourage you to follow your instincts and see where the proof breaks down. In any case, show a countermodel and counterexample.

`(∀ x : (∃ y : (p x y z))) → (∃ y : (∀ x : (p x y z)))` •

Intuitively, Question 304 tells us that if you can find a `y` that "works" for every `x`, then for every `x` you can find a `y` that works: just use the `y` that works for everyone. But Question 305 shows that the converse isn't true: knowing that for every `x` you can find a `y` that works for that `x` is not enough to establish that you can find a `y` that works for all `x`. `(∃ y : (∀ x : (p x y z)))` is stronger than `(∀ x : (∃ y : (p x y z)))`.

A summary of the "quantifier swap" theorems is that you can swap the order of adjacent quantifiers of the same kind while preserving equivalence, but if you move an existential quantifier from inside a universal to outside of it, you strengthen the formula.

• **Question 306**:

`((∀ x : (p x y)) ∧ (∃ x : (q x y))) → (∃ x : ((p x y) ∧ (q x y)))` •

• **Question 307**:

`(¬(endp x)) → (∃ e : (mem e x))` •

• **Question 308**:

`(mem e x) → (∃ i : ((natp i) ∧ (< i (len x)) ∧ ((nth i x) = e)))` •

• **Question 309**:

```
(∃ e :  (mem e x)) → (¬(endp x)) •
```

• **Question 310**:

```
((∀ j :  (((natp j) ∧ (< j (len x))) → (symbolp (nth j x))))
 ∧
 (∀ e :  ((mem e y) → (∃ i :  ((natp i) ∧ (< i (len x)) ∧ ((nth i x) = e)))))
 ∧
 (¬(endp y)))
→
 (symbolp (first y)) •
```

• **Question 311**:

```
(subeq x y) → (∀ e :  ((mem e x) → (mem e y))) •
```

• **Question 312**:

```
 (∀ e :  ((mem e x) → (mem e y)))
→
 (∀ e :  ((mem e (rest x)) → (mem e y))) •
```

• **QUESTION 313**:

```
 (∀ e :  ((mem e x) → (mem e y))) → (subeq x y)
```

Hint: Induction works with quantified formulas! Look at the proof of `perm-how-many` (page 217) for a similar problem. Also consider using the theorem of Question 312. The relevant definitions are at the end of this section. Note that this question together with Question 311 establish that

```
(subeq x y) ↔ (∀ e :  ((mem e x) → (mem e y))) •
```

• **Question 314**:

```
 (∀ x :  (f x) = (if (zp x) 1 (* x (f (- x 1)))))
→
 ((f x) = (fact x)) •
```

• **QUESTION 315**: This is a long question. Ultimately you will be asked to prove that reachability in a finite directed graph is transitive. But let me introduce the basics first. Intuitively, a "finite directed graph" is a finite collection of "nodes" connected by "arrows" or "directed arcs" from one node to another. Formally in mathematics and computer science, a *finite directed graph* is a doublet $(v\ e)$, where v is the finite set of *nodes* or *vertices* and e is the set of *edges*. We represent the "sets" of nodes and edges as lists here. An *edge* is a doublet containing the names of two nodes, the *source* and a *destination* and denotes an arc from the one to the other.

For example, the graph in Figure 7.1 could be represented by the following list of edges on the nodes `(A B C D E F G)`.

```
((A B C D E F G)               ; nodes
 ((A B) (A C) (A D)            ; edges
  (B A) (B G)
  (C D)
  (D A) (D E)
  (E D) (E G) (E F)
  (G F)))
```

Given that order does not matter, there are many other lists representing this same graph.

A *finite path*, `p`, in a graph `g` is a non-empty sequence (list) of nodes of `g` such that each adjacent pair of nodes in `p` are connected by an edge in `g` from the earlier element of `p` to the later element of `p`. A singleton list containing a single node of `g` is an empty path. The function `pathp` is defined so that `(pathp p g)` returns `t` iff `p` is a finite path in the finite directed graph `g`. All the relevant definitions are listed at the end of this section.

```
(defun pathp (p g)
  (if (endp p)
      nil
      (if (endp (rest p))
          (mem (first p) (nodes g))
          (and (mem (list (first p) (second p))
                    (edges g))
               (pathp (rest p) g)))))
```

We define `start-node` and `end-node` to return the first and last nodes in a path; see below.

For example, `(A B G F)` is a finite path from `A` to `F` in Figure 7.1. Its `start-node` is `A` and its `end-node` is `F`. Another path from `A` to `F` is `(A D A C D E F)`. Note that there may be an infinite number of finite paths from one node to another in a finite directed graph. For example, `(A B G F)`, `(A B A B G F)`, `(A B A B A B G F)`, ... are distinct paths from `A` to `F`.

Given a graph `g` and two nodes `x` and `y`, we say `y` is *reachable* from `x` if there is a path from `x` to `y` in `g`.

(reachablep y x g) ↔ (∃ p : (pathp p g) ∧ ((start-node p) = x) ∧ ((end-node p) = y))

Prove that reachability is transitive.

Theorem `reachablep-trans`:

```
((graphp g)
 ∧
 (reachablep b a g)
 ∧
 (reachablep c b g))
→
```

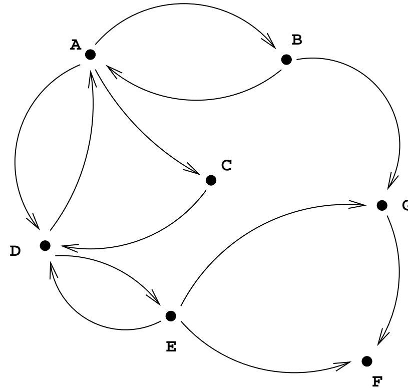


Figure 7.1: A Finite Directed Graph

```
(reachablep c a g).
```

You may provide only the first part of the proof in the following sense: reduce the theorem above to a provable formula that does not involve quantification (note that reachability involves quantification). You need not prove the resulting formula *but it must be a theorem!* •

• **Question 316**: To understand the terminology in this question, see the previous question. A path is *simple* if it does not include the same node twice. A suitable definition of `simplep` is shown among the definitions at the end of the section. If a path is not simple then it has some kind of a loop in it. Formalize and prove that if there is a path from `a` to `b` in graph `g`, then there is a simple path. The theorem will involve an existential quantifier. You should use the functions defined below, e.g., `graphp`, `pathp`, `simplep`, `start-node`, `end-node`, as necessary in your formalization. You may define additional functions if they are not included below. You may provide only the first part of the proof in the following sense: reduce your theorem to a provable formula that does not involve quantification. You need not prove the resulting formula *but it must be a theorem!* •

Here are some relevant definitions in alphabetical order.

```
(defun del1 (x y)
  (if (endp y)
      y
      (if (equal x (first y))
          (rest y)
          (cons (first y)
                (del1 x (rest y))))))


(defun edgep (edge nodes)
```

```
  (and (true-listp edge)
       (equal (len edge) 2)
       (mem (first edge) nodes)
       (mem (second edge) nodes)))


(defun edges (g) (second g))


(defun end-node (p)
  (if (endp p)
      nil
      (if (endp (rest p))
          (first p)
          (end-node (rest p)))))


(defun fact (n)
  (if (zp n)
      1
      (* n (fact (- n 1)))))


(defun graphp (g)
  (and (true-listp g)
       (equal (len g) 2)
       (list-of-edgesp (second g) (first g))))


(defun how-many (e x)
  (if (endp x)
      0
      (if (equal e (first x))
          (+ 1 (how-many e (rest x)))
          (how-many e (rest x)))))


(defun len (x)
  (if (endp x)
      0
      (+ 1 (len (rest x)))))


(defun list-of-edgesp (edges nodes)
  (if (endp edges)
```

```
      t
      (and (edgep (first edges) nodes)
           (list-of-edgesp (rest edges) nodes))))


(defun mem (x y)
  (if (endp y)
      nil
      (if (equal x (first y))
          t
          (mem x (rest y)))))


(defun nodes (g) (first g))


(defun nth (n x)
  (if (zp n)
      (first x)
      (nth (- n 1) (rest x))))


(defun pathp (p g)
  (and (not (endp p))
       (mem (first p) (nodes g))
       (or (endp (rest p))
           (and (mem (list (first p) (second p))
                     (edges g))
                (pathp (rest p) g)))))


(defun perm (x y)
  (if (endp x)
      (endp y)
      (and (mem (first x) y)
           (perm (rest x)
                 (del1 (first x) y)))))


(defun simplep (p)
  (if (endp p)
      nil
      (if (endp (rest p))
          t
          (if (mem (first p) (rest p))
```

```
              nil
              (simplep (rest p))))))


(defun start-node (p) (first p))


(defun subeq (x y)
  (if (endp x)
      t
      (and (mem (first x) y)
           (subeq (rest x) y))))
```

Set Theory (Assignment 17: 5 days)

Sets are collections of objects. Sets are themselves objects. In those two senses, sets are like lists.

Sets are different from lists in three respects:

- sets are unordered; there is no sense in which a set has a "first element," a "second," and so on;
- sets do not contain duplicates; an object is either in a set or not in the set, but it may not appear in the set multiple times; and
- sets may contain an infinite number of objects.

The basic operations on sets are familar from lists: we can

- determine if an object is an element of a set (e.g., `mem` for lists)
- determine whether one set is a subset of another (e.g., `subeq`)
- determine if two sets are equal (e.g., `equal`)
- determine the number of elements of a set (e.g., `len`)
- (possibly) add an element to a set (e.g., `cons`)
- exhibit a particular set by writing down its elements (e.g., a quoted list constant)
- create a (possibly) new set by unioning two sets (e.g., `unn`)
- create a (possibly) new set by intersecting two sets (e.g., `int`)
- create a (possibly) new set by taking the difference of one set with another (e.g. `diff`)

In fact, we can mimic finite sets with lists, by listing the elements in some canonical ordering. But if x is a list that contains all and only the elements of some finite set s I wanted to mimic, I could remove the duplicates from `x` and then sort it into `lexorder` to produce a canonical (unique) representation of the set I had in mind. That is done by `make-set` below.

```
(defun make-set (x)
```

```
(isort (rem-dups x)))
```

(See Questions 91 (page 74) and 115 (page 80).) If we use `make-set` to convert `(3 1 3 2 1 4)` into a set, we get `(1 2 3 4)`. If we convert `(1 4 3 2 2)` into a set, we also get `(1 2 3 4)`. Note that the two lists, `(3 1 3 2 1 4)` and `(1 4 3 2 2)` are different objects, i.e., they are un-`equal`. But their set counterparts are `equal`.

We could define `set-unn`, `set-int`, and `set-diff` to returns "sets," by calling `make-set` on the lists returned by `unn`, `int`, and `diff` respectively (see Questions 97 (page 75), 98 (page 75), and 99 (page 76)). If we did that, then `(set-unn '(1 4 1) '(3 2 1 2))` would be `equal` to `(set-unn '(3 2 1 2) (1 4 1))`: they would both return `(1 2 3 4)`. Thus, `set-unn` would be commutative.

But no matter what we do, we cannot represent infinite sets as lists. In set theory there is an object – a set – that contains *all* the natural numbers. There is no such list! There is another set object that contains all the conses, and we can get still another set by unioning those two sets together to get a set that contains all the natural numbers and all the conses.

Thus, while this little introduction to set theory via lists may give you an intuitive, computational idea of where we're headed, it is best to present set theory from scratch, without considering the list representation of (finite) sets.

In the rest of this chapter I'm going to mix traditional mathematical notation and ACL2. In particular, I'm going to use ACL2 notation to talk about lists because you have developed good pattern recognition for such things as `(first (cons x y))` instead of $\texttt{first}(\texttt{cons}(\texttt{x}, \texttt{y}))$. But for a function f that operates on sets I may write, $f(x, y)$ instead of `(`f` x y)`, so you can develop pattern recognition habits for traditional notation too. I'm going to use traditional notation to talk about arithmetic, but as always it is to be understood as ACL2 arithmetic. For example, I'll write "$\texttt{x} + 3 < \texttt{y}$ and mean it as "`(< (+ x 3) y)`". This mixed notation won't be as confusing as it may seem because I'll not mix the domains as much as I could.

In the following imagine that ACL2's universe (ACL2's $\mathcal{D}$) includes all the usual things – numbers, conses, etc., – plus all the sets you can build from things in the universe.[1] So I will speak, for example, of the *set* containing `23` and the list `(Mon Wed Fri)`, and will write that set as $\{$`23`, `(Mon Wed Fri)`$\}$. But understand that you cannot enter this notation into the ACL2 system. This set is different from the list `(23 (Mon Wed Fri))`.

Occasionally below I will talk about "pure ACL2 objects" and by that I mean ones composed only of the familiar ACL2 things – no sets. Suppose we define `pure-acl2-objp` so that `(pure-acl2-objp x)` is the recognizer for this condition. We actually can define this function because we have ways of recognizing every acceptable number, character, string, symbol, and cons composed of such things.[2]

Furthermore, let's give ourselves predicate *setp* and imagine that it recognizes the set objects

[1] A careful review of the ACL2 axioms will reveal that nothing in the axioms prohibits the inclusion of objects of different types than we've had so far.

[2] There is no axiom that says that every object is pure, even though those are the only ones you can type or construct in ACL2.

among all the mathematical objects in ACL2's $\mathcal{D}$. Thus, *setp*(s) is `t` if s is a set and `nil` otherwise.

Finally, let's extend the disjointness axioms of ACL2 so we know that sets are not numbers, not characters, not strings, not symbols, and not conses.

8.1 Basic Set Notation (Assignment 17 cont'd)

8.1.1 Set Builder Notation

Let v be a variable symbol and let ψ be some formula (typically involving v). Then $\{v : \psi\}$ is the set of all v satisfying ψ. It is typically read "the set of all v such that ψ." This is called *set builder notation* or sometimes *set comprehension.*

For example, $\{\texttt{x} : \texttt{(natp x)} \wedge \texttt{x} < 5\}$ is the set that contains all the natural numbers less than 5. Thus, that set contains 0, 1, 2, 3, and 4 and no other elements.

So here is an infinite set: $\{\texttt{x} : \texttt{(natp x)}\}$. Considering that this set contains an infinity of elements, this is pretty succinct notation.

Note: If you're typing your homework, I recommend using a slightly unconventional set notation comparable to the ASCII quantifier notation we've adopted. In particular, the set written in these notes as "$\{$ `x : (natp x)` $\wedge$ `(x > 7)`$\}$" should be typed in ASCII as "`(setof x (natp x) && (x > 7))`". Note that you are allowed to drop the outermost parentheses around the formula inside a `setof`-expression. Thus, you could have also written "`(setof x ((natp x) && (x > 7)))`". As usual, be extremely careful to write well-formed expressions in your homework.

The set of natural numbers is often denoted by **N** or $\mathcal{N}$ or $\mathbb{N}$.

• **QUESTION 317**: Write the set of all even natural numbers in set builder notation. You may assume we have the function `evenp` that recognizes even natural numbers. •

• **Question 318**: Write the set of all true lists of symbols. For example `(A B C)` is in it but `(A B 3)` and `(A B . C)` (the list constructed by `(cons 'A (cons 'B 'C))`) are not. •

• **QUESTION 319**: Let `f` be a function symbol of arity 2. Let's say `f` has a "pivot" at `y` if there's an `x` such that `(f x y)` = `x`. Write down the set all pivots of f. •

Warning: This notational device can get us into trouble! It is based on the assumption, called the *Axiom of Comprehension* in set theory, that to every formula of predicate calculus (containing one free variable v) there exists a set consisting of exactly the elements v satisfying that formula. This innocuous looking assumption can lead to contradictions! For example, let S be the set of all sets that don't contain themselves as an element. Is S an element of S? If it is, then according to the definition of S, S isn't an element of S. On the other hand, if S is not an element of S, then it is an element of S. So if we can talk about such "sets" as S then we have a contradiction. This is called *Russell's Paradox* after

Bertrand Russell who realized that "naive set theory" was inconsistent at the opening of the 20th century – and who helped put it on a firm foundation.

We can avoid the contradictions by settling on some fixed universe $\mathcal{D}$ and thinking of "$\{v : \psi\}$" as *really* meaning "the set of all v such that v is in $\mathcal{D}$ and ϕ holds for v." But we will largely ignore $\mathcal{D}$ here, as will most of your CS professors who use set builder notation.

It should be clear that $\{v : \psi\}$ is a kind of "quantifier" in the sense that it introduces a bound variable, v. The variable symbol after the open set brace ("{") is a *binding occurrence*. Any occurrences of v in ψ is a *bound occurrence*. The ψ in the set builder notation above is the *body*.

We extend our rules of inference so that we can do alpha-conversion on set builder notation in exactly the analogous way we do alpha-conversion on other quantified formulas.

We also extend our rules of inference so that we can rewrite and use hypotheses inside the body of a set builder expression, under the same rules regarding variables and capture. Thus, if $\psi \leftrightarrow \psi'$ then $\{x : \psi\} = \{x : \psi'\}$.

The rules for instantiating free variables in $\{v : \psi\}$ are analogous to instantiating free variables in $(\forall v : \psi)$. One should rename v to avoid capture.

8.1.2 Roster Notation

There is a special case of this notation used to write down explicit finite set constants. In it, you just write down the elements of the set, separated by commas and enclosed in set braces. Thus, the notation $\{0, 1, 2, 3, 4\}$ is the set containing just the five elements exhibited. This is called *roster* notation because you are simply giving the roster of the elements in the set.

The order in which you list them is unimportant. We could have written this same set as $\{0, 2, 4, 1, 3\}$, just like we could write `123` as `+00123`.

The notation $\{0, 1, 2, 3, 4\}$ is shorthand for $\{\mathtt{x} : (\mathtt{x} = 0) \vee (\mathtt{x} = 1) \vee (\mathtt{x} = 2) \vee (\mathtt{x} = 3) \vee (\mathtt{x} = 4)\}$.

The set $\{0, 1, 2, 3, 4\}$ is equal to the set $\{\mathtt{x} : \mathtt{(natp\ x)} \wedge \mathtt{x} < 5\}$; both contain just the five natural numbers below 5. We'll be able to prove that.

Note: When typing roster notation for your homework, you might use the syntax "`(set` e_1 e_2 `...)`" for "$\{e_1, e_2, \ldots\}$".

Sometimes people duplicate an element when using roster notation. Imagine you saw $\{0, 0, 0, 0\}$. Despite appearances, that set has only one element, 0! Sets do not have duplications.

You might wonder why somebody would write a set in roster notation and duplicate elements. The reason is that they often use expressions to denote the values of the various elements and some of those expressions may have the same value as others!

• **QUESTION 320**: Suppose j is understood to be a natural number but that you don't

know which natural number it is. Is there a way that $\{j, j+j, j*j, j-j\}$ might contain only one element? If so, explain how. •

• **QUESTION 321**: Consider the set in roster notation shown in the Question 320, page 246. Can it contain two elements? If so, explain how. •

• **QUESTION 322**: Consider the set in roster notation shown in the Question 320, page 246. Can it contain three elements? If so, explain how. •

• **QUESTION 323**: Consider the set in roster notation shown in the Question 320, page 246. Can it contain four elements? If so, explain how. •

• **Question 324**: Consider the set in roster notation shown in the Question 320, page 246. Can it contain five elements? If so, explain how. •

Don't be fooled! The number of expressions used in roster notation is not necessarily the number of elements of the set!

When we use roster notation, sometimes we write constants inside the set braces, as in $\{1, 2, 3\}$ and other times we write expressions whose values denote the elements, as in $\{j, j+j, j*j, j-j\}$. This raises a special problem if it is not always possible to distinguish objects from expressions. What set is meant by {`(first x)`}? In particular, is the element of that set the list of length 2 whose first component is the symbol `first`? Or is the element the value of the `first` component of the value of the variable `x`? When there is potential confusion we tend to use quotes. We would write {`'(first x)`} to denote the set whose element is the list object of length 2 whose first component is the symbol `first` and whose second component is the symbol `x`.

8.1.3 The Empty Set

The empty set is the set containing no elements. In roster notation, we can write the empty set as "$\{\}$". The empty set is typically denoted by the symbol "$\emptyset$" ("phi" pronounced "fi" and rhymes with "hi"). It is sometimes written "ϕ", "φ", or "Φ". We use "$\emptyset$". Thus, $\emptyset = \{\}$.

Note: These five symbols are all different: $\emptyset$, ϕ, φ, Φ, 0. The first is the symbol for the empty set, the second, third, and fourth are the Greek letter "phi" in both lower- and uppercase, and the last is the digit zero (which is sometimes written with a slash through it). To avoid confusion, I will not use the Greek letter "phi" and my zeros won't be slashed.

• **Question 325**: Write the empty set in set builder notation. •

Students often confuse the empty set with the set containing the empty set. They are different! That is, $\emptyset \neq \{\emptyset\}$.

The empty set contains 0 elements. The set containing the empty set contains 1 element. So they're different!

As a computer scientist you should have little trouble remembering that the empty set is

different from the set containing it. In ACL2, you know `nil` is different from `(nil)`. In Java you might have an empty Object of some Class and you know that an Object which contains that empty Object is different from the empty Object. The empty set is an object, a real thing. It can be in other sets. It is different from any set that contains something.

- **Question 326**: Which of these sets are the same:

1. $\emptyset$
2. $\{\emptyset\}$
3. $\{x : \texttt{(natp x)} \wedge (\texttt{x} < 0)\}$
4. $\{\emptyset, \{\emptyset\}\}$
5. $\{\emptyset, \emptyset\}$
6. $\{\}$
7. $\{\emptyset, \{\}\}$

-

8.2 Set Operations (Assignment 17 cont'd)

The basic set operations are given informally below.

name	*trad. notation*	*ASCII infix*	*description*
member	$(e \in S)$	`(`e `in` S`)`	e is an element of set S
not member	$(e \notin S)$	`(`e `!in` S`)`	e is not an element of set S
subset	$(R \subseteq S)$	`(`R `subeq` S`)`	R is a subset of set S, i.e., every element of R is in S too
equality	$(R = S)$	`(`R `=` S`)`	R is the same set as S, i.e., they have the same elements
union	$(R \cup S)$	`(`R `un` S`)`	the set containing all the elements of R together with all the elements of S
intersection	$(R \cap S)$	`(`R `int` S`)`	the set containing just the elements that R and S have in common
difference	$(R \setminus S)$	`(`R `minus` S`)`	the set obtained by removing from R the elements of S
power set	$(\wp\, S)$	`(pwr` S`)`	the set containing all the subsets of S

Because union and intersection are associative (as we shall see), you may drop the parentheses around nests of unions or nests of interesection. For example, $A \cup B \cup C$ is regarded as an abbreviation for $(A \cup (B \cup C))$.

As usual, in your homeworks take great care to type properly parenthesized well-formed formulas.

The following formal definitions of these concepts allow us to reduce most set theory conjectures to ordinary predicate calculus, which you already understand.

member	$(e \in \{v : \psi\}) \leftrightarrow (\psi/\{v \leftarrow e\})$
not member	$(e \notin S) \leftrightarrow (\neg(e \in S))$
subset	$(R \subseteq S) \leftrightarrow (\forall v : (v \in R) \rightarrow (v \in S))$
equality	$(R = S) \leftrightarrow (\forall v : (v \in R) \leftrightarrow (v \in S))$
union	$(R \cup S) = \{v : (v \in R) \vee (v \in S)\}$
intersection	$(R \cap S) = \{v : (v \in R) \wedge (v \in S)\}$
difference	$(R \setminus S) = \{v : (v \in R) \wedge (v \notin S)\}$
power set	$(\wp\, S) = \{v : v \subseteq S\}$

The first definition above is perhaps the most interesting. First, let's just make sure you understand the notation. The "$\psi/\{v \leftarrow e\}$" is the formula you get by replacing all the free occurrences of v in ψ by e. So if ψ were "`(natp x)` $\wedge$ (`x` < 5)" and e were `j`, then "$\psi/\{v \leftarrow e\}$" would be "`(natp j)` $\wedge$ (`j` < 5)." Thus, if you see

(`j` $\in$ {`x` : `(natp x)` $\wedge$ (`x` < 5)})

you could use the definition of member above to convert it to

`(natp j)` $\wedge$ (`j` < 5).

Note that this eliminates all set theory notation.

The second definition, of "not member," is just a simple abbreviation that trades "$\notin$" for a negated "$\in$". So if your goal is to get rid of set theory notation, this introduces a "$\neg$" and reduces your problem to getting rid of "$\in$".

The definition of subset eliminates "$\subseteq$" and introduces a "$\forall$", an "$\rightarrow$", and two "$\in$" expressions. So if you know how to get rid of "$\in$" you can get rid of "$\subseteq$".

You should inspect the rest of the definitions and confirm the emerging pattern: if you can get rid of "$\in$" you can get rid of all the set notation.

Also, you'll note a close correspondence between the set notation and logic: In some sense, "$\subseteq$" is just "$\rightarrow$", set equality is just "$\leftrightarrow$", "$\cup$" is "$\vee$" and "$\cap$" is "$\wedge$".

To practice using the above definitions, answer these simple questions.

• **Question 327**: Fill in the blank to make this a theorem:

$(e \in (A \cup B)) \leftrightarrow ((e \in A)__(e \in B))$ •

• **Question 328**: Fill in the blank to make this a theorem:

$(e \in (A \cap B)) \leftrightarrow ((e \in A)__(e \in B))$ •

• **Question 329**: Fill in the blank to make this a theorem:

$(e \in (A \setminus B)) \leftrightarrow ((e \in A)____(e \in B))$ •

• **QUESTION 330**: Fill in the blank to make this a theorem:

$(e \in (\wp\, A)) \leftrightarrow (e__A)$ •

But the best way to get a feel for the duality here is just to prove some theorems! In the questions below, let `P` and `Q` be Boolean functions of one argument. If an alleged "theorem" is not a theorem, show a counterexample. I urge you to read all of these problems, even those not assigned as homework, because they give you identities you will use frequently in other proofs.

• **Question 331**: $5 \in \{\mathtt{x} : \mathtt{(natp\ x)}\}$ •

• **Question 332**: $(A \cup B) = (B \cup A)$ •

My Answer: I'll prove this to illustrate the rules in action.

Proof:

$(A \cup B) = (B \cup A)$
$\leftrightarrow$ {def equality (above)}
$(\forall \mathtt{x} : (\mathtt{x} \in (A \cup B)) \leftrightarrow (\mathtt{x} \in (B \cup A)))$
$\dashv$ {∀-Concl}
$(\mathtt{x} \in (A \cup B)) \leftrightarrow (\mathtt{x} \in (B \cup A))$
$\leftrightarrow$ {union (above), twice}
$\mathtt{x} \in \{\mathtt{v} : (\mathtt{v} \in A) \vee (\mathtt{v} \in B)\} \leftrightarrow \mathtt{x} \in \{\mathtt{v} : (\mathtt{v} \in B) \vee (\mathtt{v} \in A)\}$
$\leftrightarrow$ {member (above), twice}
$((\mathtt{x} \in A) \vee (\mathtt{x} \in B)) \leftrightarrow ((\mathtt{x} \in B) \vee (\mathtt{x} \in A))$
$\leftrightarrow$ {Taut (Commutativity of Or)}
$True$
□

The combination of expanding an operation like union followed by the definition of member, as when we used two steps above to go from $(\mathtt{x} \in (A \cup B))$ to $((\mathtt{x} \in A) \vee (\mathtt{x} \in B))$ is so common we generally combine them. Also, it is frequently the case that after eliminating the quantifiers it is easier to just manipulate one side until it is identical to the other. So I might write the proof above as follows.

Proof:

$(A \cup B) = (B \cup A)$
$\leftrightarrow$ {def equality}
$(\forall \mathtt{x} : (\mathtt{x} \in (A \cup B)) \leftrightarrow (\mathtt{x} \in (B \cup A)))$
$\dashv$ {∀-Concl}

$(\mathtt{x} \in (A \cup B)) \leftrightarrow (\mathtt{x} \in (B \cup A))$
{I will transform the left-hand side to the right-hand side.}
lhs
$\leftrightarrow$
$(\mathtt{x} \in (A \cup B))$
$\leftrightarrow$ {def union and member}
$((\mathtt{x} \in A) \vee (\mathtt{x} \in B))$
$\leftrightarrow$ {Commutativity}
$((\mathtt{x} \in B) \vee (\mathtt{x} \in A))$
$\leftrightarrow$ {def union and member}
$(\mathtt{x} \in (B \cup A))$
$\leftrightarrow$
rhs
$\square$

• **Question 333**: $(A \cap B) = (B \cap A)$ •

• **Question 334**: $(A \cup (B \cup C)) = ((A \cup B) \cup C)$ •

• **Question 335**: $(A \cap (B \cap C)) = ((A \cap B) \cap C)$ •

• **QUESTION 336**: Provide a counterexample to this "identity."
$(A \cup (B \cap C)) = ((A \cap B) \cup (A \cap C))$ •

• **QUESTION 337**: Fill in the five blanks below to make this a proof. Just list the numbers 1–5 and write the contents of the corresponding blanks. Make sure your syntax is correct!

Theorem:
$(\mathtt{A} \cup (\mathtt{B} \cap \mathtt{C})) = ((\mathtt{A} \cup \mathtt{B}) \cap (\mathtt{A} \cup \mathtt{C}))$

Proof:
$(\mathtt{A} \cup (\mathtt{B} \cap \mathtt{C})) = ((\mathtt{A} \cup \mathtt{B}) \cap (\mathtt{A} \cup \mathtt{C}))$

$\leftrightarrow$ {____1____}

$(\forall\ \mathtt{e} : (\mathtt{e} \in (\mathtt{A} \cup (\mathtt{B} \cap \mathtt{C}))) \leftrightarrow (\mathtt{e} \in ((\mathtt{A} \cup \mathtt{B}) \cap (\mathtt{A} \cup \mathtt{C}))))$

$\dashv$ {∀-Concl}

$(\mathtt{e} \in (\mathtt{A} \cup (\mathtt{B} \cap \mathtt{C}))) \leftrightarrow (\mathtt{e} \in ((\mathtt{A} \cup \mathtt{B}) \cap (\mathtt{A} \cup \mathtt{C})))$

{To prove this I will reduce the left-hand side to the right-hand side.}

lhs

$\leftrightarrow$

$(\mathtt{e} \in (\mathtt{A} \cup (\mathtt{B} \cap \mathtt{C})))$

$\leftrightarrow$ {def union and member}

___2___

$\leftrightarrow$ {def intersection and member}

___3___

$\leftrightarrow$ {Distributivity}

`((e ∈ A)∨(e ∈ B)) ∧ ((e ∈ A) ∨ (e ∈ C))`

$\leftrightarrow$ {___4___ (twice)}

`(e ∈ (A ∪ B)) ∧ (e ∈ (A ∪ C))`

$\leftrightarrow$ {___5___}

`e ∈ ((A ∪ B) ∩ (A ∪ C))`

$\leftrightarrow$

rhs

□ •

Note: The last two questions illustrate an easy mistake you can make when distributing "∪" over "∩".

• **Question 338**: Which one of the following is a theorem?

1. $(A \cap (B \cup C)) = ((A \cup B) \cap (A \cup C))$
2. $(A \cap (B \cup C)) = ((A \cap B) \cup (A \cap C))$ •

- **Question 339**: $(\{\texttt{x} : \texttt{(natp x)} \land (\texttt{x} < 10)\} \cap \{\texttt{x} : \texttt{(natp x)} \land (\texttt{x} > 10)\}) = \emptyset$ •
- **Question 340**: $((A \setminus B) \cup B) = A$ •
- **Question 341**: $\{\texttt{x} : \texttt{x} \in A\} = A$ •
- **Question 342**: $(\{\texttt{x} : \texttt{(P x)}\} \cup \{\texttt{y} : \texttt{(Q y)}\}) = \{\texttt{z} : \texttt{(P z)} \lor \texttt{(Q z)}\}$ •
- **Question 343**: $(\{\texttt{x} : \texttt{(P x)}\} \cap \{\texttt{y} : \texttt{(Q y)}\}) = \{\texttt{z} : \texttt{(P z)} \land \texttt{(Q z)}\}$ •
- **Question 344**: $A \subseteq A$ •
- **Question 345**: $((A \subseteq B) \land (e \in A)) \rightarrow (e \in B)$ •
- **Question 346**: $(e \in (A \cup B)) \leftrightarrow ((e \in A) \lor (e \in B))$ •
- **Question 347**: $(e \in (A \cap B)) \leftrightarrow ((e \in A) \land (e \in B))$ •
- **Question 348**: $(e \in (A \setminus B)) \leftrightarrow ((e \in A) \land (e \notin B))$ •
- **Question 349**: $((A \subseteq B) \land (B \subseteq C)) \rightarrow (A \subseteq C)$ •
- **Question 350**: $(A = B) \leftrightarrow ((A \subseteq B) \land (B \subseteq A))$ •
- **Question 351**: $(A \subseteq (\wp\, A))$ •
- **Question 352**: $(A \in (\wp\, A))$ •
- **Question 353**: $(A \in (\wp\, (A \cup B)))$ •
- **QUESTION 354**: $(\wp\, A) \subseteq (\wp\, (A \cup B))$ •
- **Question 355**: $(A \subseteq B) \leftrightarrow ((\wp\, A) \subseteq (\wp\, B))$. •

8.3 Extensions to Quantifier and Set Builder Notation (Assignment 18: 2 days)

There are several commonly used extensions of the notation we've introduced so far and you will see them in many of your classes.

	new notation	*ASCII*	*translation*
1.	$(\forall x, y : \psi)$	`(all (x y) ψ)`	$(\forall x : (\forall y : \psi))$
2.	$(\exists x, y : \psi)$	`(exists (x y) ψ)`	$(\exists x : (\exists y : \psi))$
3.	$(\forall x \in S : \psi)$	`(all x in S ψ)`	$(\forall x : (x \in S) \rightarrow \psi)$
4.	$(\exists x \in S : \psi)$	`(exists x in S ψ)`	$(\exists x : (x \in S) \wedge \psi)$
5.	$(\forall x, y \in S : \psi)$	`(all (x y) in S ψ)`	$(\forall x \in S : (\forall y \in S : \psi))$
6.	$(\exists x, y \in S : \psi)$	`(exists (x y) in S ψ)`	$(\exists x \in S : (\exists y \in S : \psi))$
7.	$\{(f\ u) : \psi\}$	`(setof (f u) ψ)`	$\{x : (\exists u : (\psi \wedge (x = (f\ u))))\}$
8.	$\{(f\ u\ v) : \psi\}$	`(setof (f u v) ψ)`	$\{x : (\exists u, v : (\psi \wedge (x = (f\ u\ v))))\}$

Forms 1 and 2 are meant to suggest the general schema: if you see a quantifier followed by a list of variables, write the quantifier repeatedly in front of each variable.

• **Question 356**: Write the translation of $(\exists a, b, c : (a^2 + b^2) = c^2)$. •

Forms 3 and 4 are probably the most confusing for new students. The two new notations are very similar and one might expect that their translations would be the same except for the quantifier. But their translations are different! Form 3 is read, "for all x in S, ψ holds" and is translated as "for all x, if x is in S then ψ holds." But form 4 is read, "there exists an x in S such that ψ holds" and is translated "there exists an x such that x is in S and ψ holds." One might have thought the symmetry with 3 would make 4 translate as "there exists an x such that if x is in S then ψ." But if you think about it, that doesn't mean what is clearly meant by "there exists an x in S such that ψ holds."

• **Question 357**: Let S be $\{1, 2, 3\}$. If it is possible, define `psi` so that $(\exists$`x`$\in S :$ `(psi x)`$)$ is true but so that its "mis-translation" $(\exists$`x` $:$ $($`x` $\in S) \rightarrow$ `(psi x)`$)$ is false. If that is impossible, prove that when the former is true, so is the latter. •

• **Question 358**: Let S be $\{1, 2, 3\}$. If it is possible, define `psi` so that $(\exists$`x`$\in S :$ `(psi x)`$)$ is false but so that its "mis-translation" $(\exists$`x` $:$ $($`x` $\in S) \rightarrow$ `(psi x)`$)$ is true. If that is impossible, prove that when the former is false, so is the latter. •

• **Question 359**: Given the familiar fact that $(\neg(\exists x : (\neg\psi)))$ is equivalent to $(\forall x : \psi)$, you might expect that $(\neg(\exists x \in S : (\neg\psi)))$ is equivalent to $(\forall x \in S : \psi)$. Either exhibit a counterexample by showing an S and ψ or a proof of their equivalence. •

Returning to our discussion of the various forms in the table above, consider forms 5 and 6. They are meant to suggest the general schema that if you see a quantifier followed by a list of variables "in S", then write the quantifer in front of each variable and require that it be "in S."

• **Question 360**: Write the translation of $(\forall \mathtt{a}, \mathtt{b}, \mathtt{c} \in \mathbf{N} : (a^2 + b^2) = c^2)$. •

Forms 7 and 8 are meant to suggest a general schema but before I explain it, let's deal with the simpler form 7 because many students have trouble with this one.

Let `sq` be the function that squares its input and let `(evenp x)` test whether its argument is an even natural number. Then the set denoted by $\{$`(sq x)` $:$ `(evenp x)` $\wedge$ $($`x` $< 10)\}$ is $\{4, 16, 36, 64\}$, since the even naturals below 10 are 2, 4, 6 and 8.

The translation of $\{$`(sq x)` $:$ `(evenp x)` $\wedge$ $($`x` $< 10)\}$ is $\{$`z` $: \exists$`x` $:$ `(evenp x)` $\wedge$ $($`x` $< 10) \wedge ($`z` $=$ `(sq x)`$)\}$. That is, we construct the set of all `z` with the property that we can find (there exists) an even `x` below 10 and `z` is `(sq x)`.

Form 8 suggests that more than one variable can be involved in the expression producing the elements of the set. The translation existentially quantifies over each of the variables involved in the expression.

Most generally, $\{\alpha : \psi\}$, where α is a non-variable term with free variables $x_1, x_2, \ldots, x_n$, is translated to $\{z : \exists x_1, x_2, \ldots, x_n : (\psi \wedge (z = \alpha))\}$, where z is a variable different from any of the x_i.

• **Question 361**: Let S be $\{1, 2, 3\}$. Write down in roster notation the set $\{$`(list x y)` $:$ $($`x` $\in S) \wedge (($`y` $=$ `t`$) \vee ($`y` $=$ `nil`$))\}$, which we will also write as $\{\langle$`x,y`$\rangle$ $:$ $($`x` $\in S) \wedge (($`y` $=$ `t`$) \vee ($`y` $=$ `nil`$))\}$. See the next section for a discussion of pairs. •

• **Question 362**: Let S be $\{$`'(A)`, `'(B C)`, `'(D E)`$\}$ and R be the set $\{$`'(1 2 3)`, `'(4 5 6)`$\}$. Write down in roster notation the set $\{$`(app (rev x) y)` $:$ $($`x` $\in S) \wedge ($`y` $\in R)\}$. •

• **Question 363**: Fill in the blank to make this a theorem:

$(e \in \{(f\ x) : \psi\}) \leftrightarrow$ ______.

Hint: I want you to translate the new notation $\{(f\ x) : \psi\}$ to the "old" set builder notation and then apply the definition of member. •

8.4 Pair Notation (Assignment 18 cont'd)

Your CS professors will often write "$\langle x, y\rangle$," or (x, y) to denote the ordered pair whose left component is x and whose right component is y. I will use `(1 2)` to denote the constant ordered pair containing `1` and `2` as its two components. To build a pair I will use[3]

```
(defun pair (x y) (list x y)).
(defun pairp (x)
  (and (true-listp x)
       (equal (len x) 2)))
```

[3]It is standard in Lisp to construct the ordered pair $\langle x, y\rangle$ with `(cons x y)`, but because `(cons 1 2)` constructs an object that prints with a dot, namely `(1 . 2)`, I have avoided representing ordered pairs as single conses.

You may access the two components of one of these ordered pairs with the built-in functions `first` and `second`. Thus,

```
(first (pair x y)) = x
(second (pair x y)) = y.
```

I will often use the bracket notation here so you get used to it. But you should understand it as the object constructed by `pair`. This is another example of abstract syntax (e.g., "`(pair x y)`") versus concrete syntax (e.g., "$\langle x, y\rangle$").

You should understand that a key skill in mathematics and computer science is pattern matching. If nothing else, this course is training you in pattern matching! Just as you should automatically think "$A \vee B$" when you see "$(\neg A) \rightarrow B$", you should recognize the basic truths of set theory no matter what notation you see them in.

One final note about ordered pairs in set theory... Amazingly, the ordered pair $\langle x, y\rangle$, where x and y are sets, can itself be represented as a set, say c, such that you can define functions like `first` and `second` that when applied to c produce x and y respectively. But we won't do that.

Fundamentally, sets are a much more powerful "data structure" than lists. Sets can represent lists (via nested pairs) but lists cannot represent infinite sets. Most mathematics books start with set theory because all of mathematics can be formalized in it.

I rejected using set theory as the foundation of this course because I wanted to introduce logic in a computational setting.

8.5 Generalized Union and Intersection (Assignment 18 cont'd)

Suppose that S is a set of sets. Then $\bigcup S$ is the union of all the sets in S. For example, if S is $\{\{1,2,3\},\{\texttt{A},\texttt{B}\},\{\texttt{"Hi"},\texttt{"there"}\}\}$ then $\bigcup S$ is $\{1,2,3,\texttt{A},\texttt{B},\texttt{"Hi"},\texttt{"there"}\}$. More generally, if S is $\{S_1, S_2, \ldots, S_n\}$, then $\bigcup S$ is $S_1 \cup S_2 \cup \ldots \cup S_n$.

We define $\bigcap S$ analogously, using intersection instead of union. Thus if S is $\{S_1, S_2, \ldots, S_n\}$, then $\bigcap S$ is $S_1 \cap S_2 \cap \ldots \cap S_n$.

Here is how we define these new operators on sets of sets:

$$(\bigcup S) = \{x : (\exists y \in S : x \in y)\}$$
$$(\bigcap S) = \{x : (\forall y \in S : x \in y)\}$$

However, $\bigcap \emptyset$ is undefined, much like division by 0 is undefined.

Note: When writing "$\bigcup$" and "$\bigcap$" in typed homework, please use the names "`big-un`" and "`big-int`" respectively.

- **Question 364**: Why is $\bigcap \emptyset$ undefined? What would it be if it were defined?
- **QUESTION 365**: Let S be

```
{{1, 2, 3, 4},
 {1, 3, 4, 5},
 {3, 4, 5, 7},
 {3, 6, 9}}
```

Write down $\bigcup S$ in roster notation. •

• **Question 366**: Let S be the set of sets shown in the last question. Write down $\bigcap S$ in roster notation. •

• **Question 367**: Suppose A is $\{A_1, A_2, A_3\}$, where the A_i are sets. Fill in the blank so that this is a theorem.

$(e \in (\bigcup A)) \leftrightarrow ((e \in A_1)__(e \in A_2)__(e \in A_3))$ •

• **Question 368**: Suppose A is $\{A_1, A_2, A_3\}$, where the A_i are sets. Fill in the blank so that this is a theorem:

$(e \in (\bigcap A)) \leftrightarrow ((e \in A_1)__(e \in A_2)__(e \in A_3))$ •

Here is an example proof about generalized union and the power set.

Theorem. $P \subseteq (\wp\,(\bigcup P))$

Proof. I will transform the formula to true.

$P \subseteq (\wp\,(\bigcup P))$

$\leftrightarrow$ {def $\subseteq$}

$(\forall x : (x \in P) \rightarrow (x \in (\wp\,(\bigcup P))))$

$\dashv$ {$\forall$-Concl}

$(x \in P) \rightarrow (x \in (\wp\,(\bigcup P)))$

$\leftrightarrow$ {def $\wp$ and $\in$}

$(x \in P) \rightarrow (x \subseteq (\bigcup P))$

$\leftrightarrow$ {def $\subseteq$}

$(x \in P) \rightarrow (\forall y : (y \in x) \rightarrow (y \in (\bigcup P)))$

$\dashv$ {$\forall$-Concl}

$(x \in P) \to ((y \in x) \to (y \in (\bigcup P)))$

$\leftrightarrow$ {promotion}

$((x \in P) \wedge (y \in x)) \to (y \in (\bigcup P))$

$\leftrightarrow$ {def $\bigcup$}

$((x \in P) \wedge (y \in x)) \to (y \in \{e : (\exists d : (d \in P) \wedge (e \in d))\})$

$\leftrightarrow$ {def $\in$}

$((x \in P) \wedge (y \in x)) \to (\exists d : (d \in P) \wedge (y \in d))$

$\leftrightarrow$ {$\exists$-Concl}

$((x \in P) \wedge (y \in x)) \to (((x \in P) \wedge (y \in x)) \vee (\exists d : (d \in P) \wedge (y \in d)))$

$\leftrightarrow$ {Taut}

$True$

$\square$

8.6 Cross Product (Assignment 18 cont'd)

The *cross product* of two sets A and B, written $A \times B$, is the set of all ordered pairs $\langle x, y\rangle$ such that x is an element of A and y is an element of B.

We define:

$(S \times R) = \{\texttt{(pair x y)} : (\texttt{x} \in S) \wedge (\texttt{y} \in R)\}$.

Remember, you will most often see this written:

$(S \times R) = \{\langle \texttt{x}, \texttt{y}\rangle : (\texttt{x} \in S) \wedge (\texttt{y} \in R)\}$.

For example, let A be $\{\texttt{Mon}, \texttt{Wed}, \texttt{Fri}\}$ and let B be $\{1, 2, 3, 4\}$. Then $A \times B$ is

```
{ '(Mon 1), '(Mon 2), '(Mon 3), '(Mon 4),
  '(Wed 1), '(Wed 2), '(Wed 3), '(Wed 4),
  '(Fri 1), '(Fri 2), '(Fri 3), '(Fri 4) }
```

Note: When writing "×" for cross-product in your typed homework, please use the symbol

`cross` either as an infix or prefix operator, e.g., "`(A × B)`" should be written either "`(A cross B)`" or "`(cross A B)`".

• **Question 369**: Write down (in elaborated set builder notation) the expression describing the set of all pairs consisting of a natural number and its negation. Write down the same set in standard builder notation.

• **Question 370**: Prove $((A \times B) \times C) = (A \times (B \times C))$ or exhibit a counterexample. •

• **QUESTION 371**: Prove $(A \times B) = (B \times A)$ or exhibit a counterexample. •

8.7 Relations (Assignment 19: 7 days)

Informally, a "relation" is an association between objects. You are familiar with many relations, e.g., "<", "≤, and, of course, "=". But until now you have probably thought of a relation as some kind of symbol you wrote between two arithmetic quantities. But have you ever thought of a relation as an *object*? By "object" here I mean as a mathematical entity you could pass around, inspect, and manipulate.

Imagine the idea. Let $R_<$ be the object that *is* the less than relation on the natural numbers. What could you do with it? The obvious thing would be to ask whether two items, x and y are related by $R_<$. This is called "applying" $R_<$ to x and y. The result is a truth value: True if x is related to y by $R_<$ and False if they are not. For example, if we applied $R_<$ to `1` and `3`, we would get True. If we applied $R_<$ to `3` and `1` we would get False. But if $R_<$ is an object we could do other things with it. We could pass it around to functions. For example, we could write a general-purpose sorting algorithm that takes two arguments, a list of naturals and a relation, and sorts the list into the order specified by the relation. If relations were objects we could write functions that manipulated them, for example that combined two relations to get a new relation, etc.

There are many examples of relations besides the familiar arithmetical ones. For example, a graph is a relation.

In the finite directed graph of Figure 8.1, the arrows from `A` to `B`, `C`, and `D`, indicate that `A` is related to each of those nodes. Graphs are just examples of relations. Examples of graphs abound in computer science. The most well known graph is the Internet, where each host is connected by edges to neighboring hosts and messages are transferred from sender to receiver by routing protocols that cause each message to hop from host to host. Of course, the Internet is a graph or relation between millions of items (hosts).

We have shown how such graphs or relations can be represented formally as lists of pairs (see page 236). Because lists have to be finite, such a representation of graphs would force our graphs to have a finite number of nodes. Even the Internet has a finite number of hosts, so mathematically lists would suffice to represent it.

But if we wanted to represent a relation on an infinite number of objects, we could not do it with lists. Sets do not have to be finite. Therefore, computer scientists and mathematicians

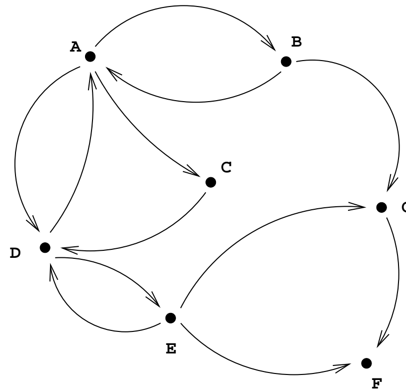


Figure 8.1: A Finite Directed Graph

use sets rather than lists to formalize relations. The graph of Figure 8.1 could be formalized as the set

{'(A B), '(A C), '(A D), '(B A), '(B G),
 '(C D), '(D A), '(D E), '(E D), '(E G), '(E F), '(G F)}.

Each pair, (x y), in the relation indicates that its first, x, is related to its second, y.

An example of an infinite relation is, of course, "<". Here are some of the pairs in the "<" relation $R_<$.

```
R< = {'(0 1), '(0 2), '(0 3), '(0 4), ...
              '(1 2), '(1 3), '(1 4), ...
                      '(2 3), '(2 4), ...
                              ...
                              }
```

Of course, we don't write infinite sets in roster notation! Instead, we use set builder notation:

$R_< = \{\langle \texttt{x}, \texttt{y} \rangle : \texttt{(natp x)} \wedge \texttt{(natp y)} \wedge (\texttt{x} < \texttt{y})\}$

You should recognize that the set builder notation above describes the infinite set alluded to in the incomplete attempt to write $R_<$ in roster notation.

Of course, we can have infinite relations over things other than numbers. Here is the "same `len`" relation over pure ACL2 objects, where $\mathcal{P} = \{\texttt{x} : \texttt{(pure-acl2-objectp x)}\}$.

$R_{samelen} = \{\langle \texttt{x}, \texttt{y} \rangle \in (\mathcal{P} \times \mathcal{P}) : \texttt{(len x)} = \texttt{(len y)}\}$

In this relation, `(A B C)` is related to `((1) Monday 7)` and also to `(1 2 3 . 45)` because they all have the same `len`.

A *relation* is just a set of ordered pairs.

$Relation(R) \leftrightarrow (\forall \texttt{x} \in R : \texttt{(pairp x)})$

• **Question 372**: Let us say two things "overlap" if and only if they are sets with an element in common. Define the overlap relation. •

R is a *binary relation on set* A iff R is a relation and a subset of the cross-product of A with itself.

There are many notations to express the idea that `x` and `y` are in relation R. Among them are

- ♦ `(pair` x y`)` $\in R$
- ♦ $\langle x, y \rangle \in R$,
- ♦ `(`R `x y)`,
- ♦ $R(\texttt{x}, \texttt{y})$,
- ♦ Rxy, and
- ♦ xRy.

The *domain* of a relation R, written $dom(R)$, is the set of all the first components of the pairs in R and the *range*, written $ran(R)$, is the set of all the second components. For example, for the relation

{'(0 ZERO), '(1 POS), '(2 POS), '(-1 NEG), '(-2 NEG)}

the domain is $\{\texttt{-1}, \texttt{-2}, \texttt{0}, \texttt{1}, \texttt{2}\}$ and the range is $\{\texttt{NEG}, \texttt{ZERO}, \texttt{POS}\}$. That is, x is in the domain of relation R iff there is some y such that `(pair` x y`)` is in R. Symmetrically, y is in the range of R iff there's some x such that `(pair` x y`)` is in R. Put still another way, x is in the domain of R if you can find a y such that xRy, and symmetrically for the range.

• **Question 373**: Define formally the concepts of *dom* and *ran*. You may assume R is a relation. •

• **Question 374**: The *converse* of relation R is the relation you get by swapping each pair in R. The converse of R is often written R^{-1}. If R is the relation

{'(0 ZERO), '(1 POS). '(2 POS), '(-1 NEG), '(-2 NEG)}

then R^{-1} is

{'(ZERO 0), '(POS 1). '(POS 2), '(NEG -1), '(NEG -2)}

Define the converse formally. •

• **Question 375**: Let R be a relation with domain A and range B. Is R equal to $A \times B$? If not, show an R, A, and B, that are a counterexample. •

• **Question 376**: Let R be a relation with domain A and range B. What is the domain of R^{-1}? What is the range of R^{-1}? •

• **Question 377**: Let R be a relation on A. Is the domain of R equal to A? If not, show an R and A that are a counterexample. •

• **QUESTION 378**: Let R be a relation on A. Is the range of R equal to A? If not, show an R and A that are a counterexample. •

Relations have many important properties. You should learn all of these! In the following, we assume R is a relation on A.

- ♦ R is *reflexive* (on A): $(\forall \mathtt{x} \in A : (R\ \mathtt{x}\ \mathtt{x}))$
- ♦ R *irreflexive* (on A) : $(\forall \mathtt{x} \in A : (\neg(R\ \mathtt{x}\ \mathtt{x})))$
- ♦ R is *symmetric* (on A): $(\forall \mathtt{x}, \mathtt{y} \in A : (R\ \mathtt{x}\ \mathtt{y}) \to (R\ \mathtt{y}\ \mathtt{x}))$
- ♦ R is *asymmetric* (on A): $(\forall \mathtt{x}, \mathtt{y} \in A : (R\ \mathtt{x}\ \mathtt{y}) \to (\neg(R\ \mathtt{y}\ \mathtt{x})))$
- ♦ R is *antisymmetric* (on A): $(\forall \mathtt{x}, \mathtt{y} \in A : ((R\ \mathtt{x}\ \mathtt{y}) \wedge (R\ \mathtt{y}\ \mathtt{x})) \to \mathtt{x} = \mathtt{y})$
- ♦ R is *transitive* (on A): $(\forall \mathtt{x}, \mathtt{y}, \mathtt{z} \in A : ((R\ \mathtt{x}\ \mathtt{y}) \wedge (R\ \mathtt{y}\ \mathtt{z})) \to (R\ \mathtt{x}\ \mathtt{z}))$
- ♦ R is *total* (or *strongly connected*) (on A): $(\forall \mathtt{x}, \mathtt{y} \in A : (R\ \mathtt{x}\ \mathtt{y}) \vee (R\ \mathtt{y}\ \mathtt{x}))$
- ♦ R is *connected* (on A): $(\forall \mathtt{x}, \mathtt{y} \in A : (R\ \mathtt{x}\ \mathtt{y}) \vee (R\ \mathtt{y}\ \mathtt{x}) \vee (\mathtt{x} = \mathtt{y}))$

• **Question 379**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is reflexive on A. •

• **Question 380**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is irreflexive on A. •

• **Question 381**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is symmetric on A. •

• **Question 382**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is asymmetric on A. •

• **Question 383**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is antisymmetric on A. •

• **Question 384**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is transitive on A. •

• **Question 385**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is

strongly connected on A. •

• **Question 386**: Let A be $\{0, 1, 2\}$. Write down in roster notation a relation R that is connected on A. •

• **QUESTION 387**: Let

$$R_{=} = \{\langle x, y\rangle : (x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x = y)\}.$$
$$R_{\neq} = \{\langle x, y\rangle : (x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x \neq y)\}.$$
$$R_{<} = \{\langle x, y\rangle : (x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x < y)\}.$$
$$R_{\leq} = \{\langle x, y\rangle : (x \in \mathbf{N}) \wedge (y \in \mathbf{N}) \wedge (x \leq y)\}.$$

For each cell in the table below, either indicate that the given relation has the given property on **N**, or provide a counterexample.

property	$R_{=}$	$R_{\neq}$	$R_{<}$	$R_{\leq}$
reflexive				
irreflexive				
symmetric				
asymmetric				
antisymmetric				
transitive				
strongly conn.				
connected				

•

• **Question 388**: Prove that if a relation R on A is reflexive and connected then it is strongly connected. •

8.8 Orders (Assignment 19 cont'd)

Suppose R is a relation on A.

Then R is a *partial order on* A iff R is reflexive on A, antisymmetric on A, and transitive on A.

In the next few questions, we let C be the set $\{\texttt{RED}, \texttt{BLUE}, \texttt{GREEN}, \texttt{GOLD}\}$.

• **Question 389**: Write in roster notation a partial order on set C above. •

R is a *total order* (sometimes called a *weak (or non-strict) simple order* or *linear order*) on A iff R is strongly connected on A, antisymmetric on A, and transitive on A.

`Lexorder`, i.e., the set { `(pair x y)` : `(lexorder x y)`}, is an example of a total order on the set $\mathcal{P}$ of pure ACL2 objects. This follows from the following theorems about `lexorder`, where $P = \{$`x` : `(pure-acl2-objectp x)`$\}$.

strongly connected:

(∀ x,y ∈ $\mathcal{P}$: (lexorder x y) ∨ (lexorder y x))
antisymmetric:
(∀ x,y∈ $\mathcal{P}$: ((lexorder x y) ∧ (lexorder y x)) → x=y)
transitive:
(∀ x,y,z∈ $\mathcal{P}$: ((lexorder x y) ∧ (lexorder y z)) → (lexorder x z))

• **Question 390**: Write in roster notation a weak simple order on C above. •

• **Question 391**: Write in roster notation a partial order that is not a total order on C above. •

• **QUESTION 392**: Prove that every total order is a partial order. •

R is a *strong simple order on* A iff R is connected on A, asymmetric on A, and transitive on A.

• **Question 393**: Is a strong simple order a weak simple order? If so, prove it. If not, show an example of a strong simple order that is not a weak simple order. •

• **QUESTION 394**: Let A be a set of sets. Is "$\subseteq$" a partial order on A? If so, prove it. If not, show a counterexample. •

• **Question 395**: Name a weak simple order on the naturals. •

• **Question 396**: Name a strong simple order on the naturals. •

8.9 Equivalence Relations (Assignment 19 cont'd)

R is an *equivalence relation on* A iff R is reflexive on A, symmetric on A, and transitive on A.

Let $\mathcal{P}$ be the pure objects in the ACL2 universe, $\{\mathtt{x} : \mathtt{(pure\text{-}acl2\text{-}objectp\ x)}\}$.

• **Question 397**: Let $R_{samelen}$ be

$R_{samelen} = \{\langle \mathtt{x}, \mathtt{y}\rangle \in (\mathcal{P} \times \mathcal{P}) : \mathtt{(len\ x)} = \mathtt{(len\ y)}\}$.

Thus, for example, $R_{samelen}$ holds on `ABC` and `nil` (because they both have `len 0`) and on `(A B C)` and `(1 2 3)` (because they both have `len 3`). Is $R_{samelen}$ an equivalence relation on $\mathcal{P}$? If so, prove it; if not, show a counterexample. •

• **Question 398**: Prove that $\{\langle \mathtt{x}, \mathtt{y}\rangle \in (\mathcal{P} \times \mathcal{P}) : \mathtt{(perm\ x\ y)}\}$ is an equivalence relation on $\mathcal{P}$. (You may assume the theorems of Section 5.3 have been proved over $\mathcal{P}$.) •

If R is an equivalence relation on a set A and $u \in A$, then we define the *equivalence class* of u under R to be $[u]_R$, where:

$$[u]_R = \{\mathtt{x} \in A : (R\mathtt{x}u)\}.$$

Note: When typing this notation in your homeowrk, write "`(eqclass` u R`)`" for "$[u]_R$".

Often, when only one equivalence relation is being discussed, we drop the subscript R specifying the equivalence relation.

• **Question 399**: Recall `perm` from Question 109, page 78. `Perm` is an equivalence relation on $\mathcal{P}$. What is $[\texttt{(1 2 3)}]_{\texttt{perm}}$? •

By definition, every element of $[u]_R$ is equivalent (in the R sense) to u.

• **Question 400**: A "score sheet" is a relation between a finite set of names and natural numbers from 0 to 100. If we represent a score sheet as a list, an example might be:

```
((Abigail  75)
 (Anthony  72)
 (Ava      98)
 (Daniel   95)
 (Emily    81)
 (Emma     75)
 (Ethan    89)
 (Hannah   91)
 (Jacob    99)
 (Joshua   82)
 (Madison  85)
 (Matthew  70)
 (Olivia   93)
 (William  77)).
```

Let x be some student named in the score sheet and let n be the natural number related to x. Letter grades are assigned according to scores as follows

$90 \leq n \leq 100$	`A`
$80 \leq n < 90$	`B`
$70 \leq n < 80$	`C`
$60 \leq n < 70$	`D`
$0 \leq n < 60$	`F`

Two students are considered "grade equivalent" if they have the same letter grade. What grade does `Olivia` have? Define `grade-eqv` so that it takes two student names and a score sheet and returns `t` or `nil` to indicate whether the two students are grade equivalent under the score sheet. •

• **Question 401**: Following up on Question 400, page 266, prove that `grade-eqv` is an equivalence relation on the set of students named in the grade sheet. (Formally, equivalence relations take only two arguments but it is not unusual in practice to see extra arguments as here. Figure out what it means for this function to be an "equivalence relation.") **Hint**: Prove that it is an equivalence relation on the set of all pure ACL2 objects. •

• **QUESTION 402**: With respect to the grade sheet shown in Question 400, page 266 and

the equivalence relation `grade-eqv`, what what is $[\texttt{Olivia}]_{\texttt{grade-eqv}}$?

As you can see, we deal with equivalence classes all the time. As individuals, `Emily` and `Madison` may be quite different, but there is a familiar sense in which they're "the same" and that is "they each have a `B` on the score sheet." When employers look at transcripts they effectively only see equivalence classes of students, i.e., all the students who made the same grade a treated equivalently. (That is why employers often want letters of reference, to distinguish otherwise indistiguishable individuals.)

8.10 Uniqueness Notation (Assignment 20: 5 days)

How do you write a set theory formula that formalizes the idea "there is exactly one `x` such that `(p x)`"?

This is such a common utterance that there is a special notation for it: $(\exists!\texttt{x} : \texttt{(p x)})$.

The formal version of this utterance is:

$$(\exists!\ \texttt{x} : \texttt{(p x)}) \leftrightarrow ((\exists\ \texttt{x} :\ \texttt{(p x)}) \wedge (\forall\ \texttt{x,y} :\ ((\texttt{p x}) \wedge (\texttt{p y})) \rightarrow \texttt{x} = \texttt{y})).$$

That is, $(\exists!\texttt{x} : \texttt{(p x)})$ means there is an `x` with property `p` and if both `x` and `y` have property `p`, then they're the same.

8.11 Partitions (Assignment 20 cont'd)

Two sets are *disjoint* if their intersection is empty. Thus, $\{1,2,3\}$ is disjoint from $\{4,5\}$ but not from $\{2,4,6\}$.

Note: Many people characterize disjointness informally by saying "the two sets have no intersection." This is sloppy. Any two sets have an intersection! Sets are disjoint when their intersection is empty. This is like saying of a baseball game "There's no score yet." That's wrong. Of course there's a score! They just mean "the score is 0 to 0."

A "partition" of a set A is a set that shows how to divide the elements of A into disjoint subsets. For example, one partition of the set $\{0,1,2,3,4,5,6\}$ is the set $P = \{\{0,2,5\},\{1,4,6\},\{3\}\}$. No *cell* in the partition P is empty, the union of all the cells is A, and no element of A is in two cells.

P is a *partition* of A, denoted $(\texttt{Partition}\ P\ A)$ iff

$(\emptyset \notin P)$
$\wedge$
$((\bigcup P) = A)$
$\wedge$
$(\forall \texttt{a}, \texttt{b} \in P : ((\texttt{a} \cap \texttt{b}) \neq \emptyset) \rightarrow (\texttt{a} = \texttt{b}))$.

This is an incredibly succinct definition – and is typical of the use of set theory in computer science.

• **Question 403**: Write one or more English sentences explaining how each conjunct in the formal definition of "partition" formalizes part of the informal definition given above. That is, explain how the mathematical notation captures the informal ideas. •

• **Question 404**: Formalize and prove: if P is a partition of A and $a \in P$ then $a \subseteq A$ •

• **Question 405**: Is $\{\{1, 2, 3\}\}$ a partition of $\{1, 2, 3\}$? •

• **Question 406**: Write down three different partitions of the set $\{1, 2, 3\}$.

• **Question 407**: Define the ACL2 function `all-partitions` so that if given a duplication-free list representing a finite set x it returns a list representing the set of all partitions of x. For example, `(all-partitions '(1 2 3))` should return something like:

```
(((3 2 1))
 ((3 2) (1))
 ((3) (2 1))
 ((3 1) (2))
 ((3) (2) (1))).
```

I say "something like" because I want you to treat the elements like sets. For example, this is an equally valid answer:

```
(((1 2 3))
 ((1) (2 3))
 ((1 2) (3))
 ((1 3) (2))
 ((1) (2) (3))).
```

If we had true sets in ACL2, we'd say

```
(all-partitions '{1,2,3}) ⟹
{{{3, 2, 1}}
 {{3, 2} {1}}
 {{3} {2, 1}}
 {{3, 1} {2}}
 {{3} {2} {1}}}
```

and there would only be one correct answer. This is a good exercise in list programming. •

8.12 Equivalence Relations *v* Partitions (Assignment 20 cont'd)

An equivalence relation R on a set A generates (or *induces*) a partition of A: just clump together all the elements that are equivalent under R.

• **Question 408**: Recall the equivalence relation `perm` from Question 109, page 78. Let A be the set

```
{'(a b),
 '(a b c),
 '(a b d),
 '(b a),
 '(b c a),
 '(b d a),
 '(b a c),
 '(d a b),
 '(b a d)}.
```

Since `perm` is an equivalence relation on all ACL2 objects, it is an equivalence relation on A. Write down a partition A induced by `perm`. •

If R is an equivalence relation on A, then $(\mathcal{EC}\ R\ A)$, the *equivalence classes induced* by R on A, is defined as $(\mathcal{EC}\ R\ A) = \{[c]_R : c \in A\}$.

In this section, we assume R is an equivalence relation on A.

We make several observations about the equivalence classes induced by R on A. First, note that if A is non-empty, then $(\mathcal{EC}\ R\ A)$ is non-empty because there is at least one element, $c \in A$, and thus $[c]_R \in (\mathcal{EC}\ R\ A)$. Now let $a \in (\mathcal{EC}\ R\ A)$. Our second observation is that a is non-empty, because $a = [c]_R$, for some c and $[c]_R$ is non-empty because R is reflexive. Third, $a \subseteq A$, because every element of $[c]_R$ is an element of A. Fourth, $(\exists \mathtt{x} \in A : a = [x]_R)$.

Theorem: The $(\mathcal{EC}\ R\ A)$ is a partition of A (provided R is an equivalence relation on A).

Proof: We must establish three things.

(1) $\emptyset \notin (\mathcal{EC}\ R\ A)$. This follows from our second observation.

(2) $(\bigcup(\mathcal{EC}\ R\ A)) = A$. We prove this by proving that each side is a subset of the other.

(2.1) $(\bigcup(\mathcal{EC}\ R\ A)) \subseteq A$ is easy since each element of $(\mathcal{EC}\ R\ A)$ is a subset of A by our third observation.

(2.2) $A \subseteq (\bigcup(\mathcal{EC}\ R\ A))$ is proved by showing that if $(e \in A)$ then $(e \in (\bigcup(\mathcal{EC}\ R\ A)))$. The latter is equivalent to $(\exists \mathtt{c} : (\mathtt{c} \in (\mathcal{EC}\ R\ A)) \wedge (e \in \mathtt{c}))$. To prove that such a `c` exists, consider $[e]_R$.

(3) $(\forall \mathtt{a}, \mathtt{b} \in (\mathcal{EC}\ R\ A) : ((\mathtt{a} \cap \mathtt{b}) \neq \emptyset) \rightarrow (\mathtt{a} = \mathtt{b}))$. So let $\mathtt{a} \in (\mathcal{EC}\ R\ A)$ and $\mathtt{b} \in (\mathcal{EC}\ R\ A)$ and, since $(\mathtt{a} \cap \mathtt{b}) \neq \emptyset$, suppose e is an element in both `a` and `b`. We must show $\mathtt{a} = \mathtt{b}$. We do it by showing that if $d \in \mathtt{a}$ then $d \in \mathtt{b}$; the symmetric case is analogous.

By our fourth observation, $a = [u]_R$ and $b = [v]_R$, where $u, v \in A$. Since $d \in \mathtt{a}$, $(R\ u\ d)$.

Since $e \in$ `a` and $e \in$ `b`, $(R\ u\ e)$ and $(R\ v\ e)$. Thus, $(R\ v\ d)$ by the symmetry and transitivity of R. But $(R\ v\ d)$ implies $d \in$ `b`.

□

Thus, we see that the equivalence classes induced by R on A is a partition of A. Every equivalence relation induces a partition.

• **Question 409**: Formally state and prove our first observation about $(\mathcal{EC}\ R\ A)$.

• **Question 410**: Formally state and prove our second observation about $(\mathcal{EC}\ R\ A)$.

• **Question 411**: Formally state and prove our third observation about $(\mathcal{EC}\ R\ A)$.

• **Question 412**: Formally state and prove our fourth observation about $(\mathcal{EC}\ R\ A)$.

• **Question 413**: Give a more careful symbolic proof of step (1) of the Theorem above. You may use the formal statements derived in Questions 409–412 as lemmas.

• **Question 414**: Give a more careful symbolic proof of step (2) of the Theorem above. You may use the formal statements derived in Questions 409–412 as lemmas.

• **Question 415**: Give a more careful symbolic proof of step (3) of the Theorem above. You may use the formal statements derived in Questions 409–412 as lemmas.

It turns out that every partition generates an equivalence relation, but we do not prove that here.

8.13 Functions (Assignment 20 cont'd)

We have been dealing with functions since the beginning of the semester. We speak of

```
(defun len (x)
  (if (endp x)
      0
      (+ 1 (len (rest x)))))
```

as defining a function, which we call `len`. But what is `len`? What sort of mathematical object is `len`?

A similar question was answered implicitly when we spoke of `<` being a relation. In set theory, the "less than" relation on the naturals is the set of ordered pairs $\{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in \mathbf{N}) \wedge (\mathtt{y} \in \mathbf{N}) \wedge \mathtt{(< x\ y)}\}$.

In the same sense, the function `len` on the pure ACL2 objects $\mathcal{P}$ is $\{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in \mathcal{P}) \wedge (\mathtt{y} = \mathtt{(len\ x)})\}$.

That is, a function is a set of ordered pairs and in each pair the second component is the

value of the function on the first component.[4]

• **Question 416**: The set $f = \{\langle 1,2\rangle, \langle 2,3\rangle, \langle 3,0\rangle\}$ is a function. What is $f(1)$? What is $f(3)$? •

In the question above, the function f is not defined on 4. That is, 4 is not in the "domain" of the function. ACL2 functions are interesting because they are defined on all ACL2 objects. For example, you can't introduce an ACL2 function that is not defined on 4.

But not all sets are functions. Not even all sets of ordered pairs are functions. Could $f = \{\langle 1,2\rangle, \langle 1,3\rangle, \langle 3,0\rangle\}$ be a function? No! The reason is that f is not uniquely defined on 1. Functions are "deterministic" in the sense that their output is dependent only on their input and they return exactly the same thing every time they're called on a given input.

A set f is a *function*, written $Function(f)$, is defined to mean:

$$(Function f) \leftrightarrow (Relation f) \wedge (\forall \mathtt{x}, \mathtt{u}, \mathtt{w} : (((\langle \mathtt{x}, \mathtt{u}\rangle \in f) \wedge (\langle \mathtt{x}, \mathtt{w}\rangle \in f)) \rightarrow (\mathtt{u} = \mathtt{w})).$$

• **Question 417**: Is $f = \{\langle \mathtt{t}, \mathtt{nil}\rangle, \langle \mathtt{nil}, \mathtt{t}\rangle\}$ a function? If not, explain why. •

• **Question 418**: Is $f = \emptyset$ a function? If not, explain why. •

• **Question 419**: Is $f = \{\langle 1, \mathtt{t}\rangle, \langle 1, \mathtt{nil}\rangle\}$ a function? If not, explain why. •

A function f on A is the *identity* function (on A) if $(\forall \mathtt{x} \in A : f(\mathtt{x}) = \mathtt{x})$.

• **Question 420**: Write in set notation the identity function on $\mathbf{N}$. •

Recall that we defined domain and range, *dom* and *ran*, for relations. Since functions are relations, the concepts of domain and range apply to functions as well.

• **Question 421**: What is $(dom\ f)$ where f is the function

$$\{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in \mathbf{N}) \wedge (\mathtt{y} = \mathtt{(*\ 2\ x)})\}?$$

•

• **Question 422**: What is $(f\ 123)$ where f is the function above? •

• **Question 423**: What is $(ran\ f)$ where f is the function above? •

It is convenient to introduce the following notation.

$$(f : A \rightarrow B) \leftrightarrow ((Function\ f) \wedge ((dom\ f) = A) \wedge ((ran\ f) \subseteq B)).$$

We read "$(f : A \rightarrow B)$" as "f maps A to B."

• **Question 424**: Suppose $(f : \mathbf{N} \rightarrow \mathbf{N})$. Is $(ran\ f) = \mathbf{N}$? •

---

[4] To permit the representation of functions of more than one argument a more general scheme must be adopted in which the first component of each pair in a function is actually a list of all the arguments. Thus, for example, a typical pair in the set representation of the function + is $\langle \mathtt{(3\ 5)}, \mathtt{8}\rangle$. But we are going to keep things simple by working only with functions of one argument represented as sets

The "$\rightarrow$" symbol is thus used with two different meanings. We use it for logical implication and we use it to denote maps. Sometimes "$(f : A \rightarrow B)$" is called a *signature* for f because it tells us what f "expects" as input and what it delivers as output.

If f is a function, then $(f\ x)$ is the y such that $\langle x, y\rangle \in f$. If x is not in the domain of f then $(f\ x)$ is not defined.

If f is a function and $B \subseteq (dom\ f)$ then we define the *image* of f on B,

$$(Img\ f\ B) = \{\mathtt{y} : (\exists \mathtt{x} \in B : (\mathtt{y} = f(\mathtt{x})))\}.$$

We define f is *onto* B to mean

$$(onto\ f\ B) \leftrightarrow ((Function\ f) \wedge ((ran\ f) = B)).$$

We define f to be *1:1* (pronounced "one-to-one" or sometimes just "one one") as

$$(1:1\ f) \leftrightarrow ((Function\ f) \wedge (\forall \mathtt{x}, \mathtt{y}, \mathtt{z} : ((((\langle \mathtt{x}, \mathtt{y}\rangle \in f) \wedge (\langle \mathtt{z}, \mathtt{y}\rangle \in f)) \rightarrow (\mathtt{x} = \mathtt{z})))).$$

Recall the definition of the converse, R^{-1} of a relation R, obtained by swapping the first and second components of the pairs in the relation.

Theorem If f is a function, then f^{-1} is a function iff f is 1:1.

Proof: Assume f is a function.

$(1:1\ f)$
$\leftrightarrow$ { by definition of 1:1}
$(\forall \mathtt{x}, \mathtt{y}, \mathtt{z} : ((((\langle \mathtt{y}, \mathtt{x}\rangle \in f) \wedge (\langle \mathtt{z}, \mathtt{x}\rangle \in f)) \rightarrow (\mathtt{y} = \mathtt{z})))$
$\leftrightarrow$ { by definition of f^{-1}}
$(\forall \mathtt{x}, \mathtt{y}, \mathtt{z} : ((((\langle \mathtt{x}, \mathtt{y}\rangle \in f^{-1}) \wedge (\langle \mathtt{x}, \mathtt{z}\rangle \in f^{-1})) \rightarrow (\mathtt{y} = \mathtt{z})))$
$\leftrightarrow$ { by definition of *Function*}
$(Function\ f^{-1})$
$\square$

If f and g are functions and $(ran\ g) \subseteq (dom\ f)$, then the *composition* of f and g is defined:

$$f \circ g = \{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in dom(g)) \wedge (\mathtt{y} = (f\ (g\ x)))\}.$$

Note: There is some confusion about how "$f \circ g$" is read. Some people say "f composed with g" while others say "g composed with f." I won't use either of those readings; I'll just say "f circle g." Just remember how $f \circ g$ is defined and when other people use the word "compose" make sure you know which way they mean it.

If f and g are functions and $(ran\ g) \subseteq (dom\ f)$, then

1. $(Function\ (f \circ g))$, i.e., $(f \circ g)$ is a function
2. $(dom\ (f \circ g)) = (dom\ g)$

3. $(x \in (dom\ g)) \rightarrow (((f \circ g)\ x) = (f\ (g\ x))$
4. $(ran\ (f \circ g)) \subseteq (ran\ f)$.

Note that function composition is not usually commutative.

• **Question 425**: Define functions f and g such that $(f \circ g) \neq (g \circ f)$. •

However, under certain circumstances, composition is associative.

Theorem If f, g, and h are functions and $(ran\ g) \subseteq (dom\ f)$ and $(ran\ h) \subseteq (dom\ g)$, then $(f \circ (g \circ h)) = ((f \circ g) \circ h)$.

Proof:

Both sides are functions with the same domain as f. We simply show that for all $\mathtt{x}$ in $(dom\ f)$, applying the two functions yield the same thing:

$((f \circ (g \circ h))\ \mathtt{x})$
$=$
$(f\ ((g \circ h)\ \mathtt{x}))$
$=$
$(f\ (g\ (h\ \mathtt{x})))$
$=$
$((f \circ g)\ (h\ \mathtt{x}))$
$=$
$(((f \circ g) \circ h)\ \mathtt{x})$.
□

The "proof" above is really just a sketch. To complete it we have to make sure that $(ran\ (g \circ h)) \subseteq (dom\ f)$ and $(ran\ h) \subseteq (dom\ (f \circ g))$. But both follow immediately from our hypotheses and obvious facts about composition.

□

• **Question 426**: Give proofs of the two facts mentioned at the end of the proof above. •

Theorem. If f is a function and 1:1, then $(f^{-1} \circ f)$ is the identity function on $(dom\ f)$.

Proof:

Assume f is a 1:1 function. Thus, f^{-1} is a function.

Then $(f^{-1} \circ f)$ is, by the definition of composition,

$$\{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in (dom\ f)) \wedge (\mathtt{y} = (f^{-1}\ (f\ \mathtt{x})))\}$$

which is a function whose domain is $(dom\ f)$. Suppose $\mathtt{x} \in (dom\ f)$. Then $\langle \mathtt{x}, f(\mathtt{x})\rangle \in f$. Hence, by definition of converse, $\langle (f\ \mathtt{x}), \mathtt{x}\rangle \in f^{-1}$. Thus, $(f^{-1}\ (f\ x)) = x$. Thus, $(f^{-1} \circ f) = \{\langle \mathtt{x}, \mathtt{y}\rangle : (\mathtt{x} \in (dom\ f)) \wedge (\mathtt{y} = \mathtt{x})\}$, which is the identity function on $(dom\ f)$.

□

Suppose that $f : A \to B$. Then h is an *inverse* of f iff

$$(h : (Img\ f\ A) \to A) \wedge (\forall \mathtt{x} \in A : ((h \circ f)\ \mathtt{x}) = \mathtt{x}).$$

Theorem. If the function f is not 1:1, it has no inverse.

Proof:

Suppose that f is a function that is not 1:1. Thus, there exists $\mathtt{x}$ and $\mathtt{y}$, both in $(dom\ f)$ such that $\mathtt{x} \neq \mathtt{y}$ and $(f\ \mathtt{x}) = (f\ \mathtt{y})$. Suppose there is a function g that is the inverse of f. Then $(g\ (f\ \mathtt{x})) = (g\ (f\ \mathtt{y}))$. But then $\mathtt{x} = \mathtt{y}$. Contradiction.

□

Theorem. If g is an inverse of f, then g is f^{-1}.

Proof: By definition, both f^{-1} and g are functions with domain equal to the range of f. Suppose that $\mathtt{x} \in (dom\ f)$. To show that $f^{-1} = g$ it is sufficient to prove that the functions f^{-1} and g agree on $(f\ \mathtt{x})$. We know that $\langle (f\ \mathtt{x}), \mathtt{x} \rangle \in f^{-1}$, since $\langle \mathtt{x}, (f\ \mathtt{x}) \rangle \in f$. Suppose that $\langle (f\ \mathtt{x}), \mathtt{y} \rangle \in g$ and $\mathtt{x} \neq \mathtt{y}$. But then $(g\ (f\ \mathtt{x})) = \mathtt{y} \neq \mathtt{x}$, which contradicts the hypothesis that g is an inverse of f. Hence, g and f^{-1} are everywhere equal, and hence are identical.

□

8.14 Cardinality (Assignment 20 cont'd)

The cardinality $|A|$ of a finite set, A, is the number of elements in it.

Thus, $|\{2, 4, 6\}| = 3$.

• **Question 427**: What is $|\{\mathtt{x} \in \mathbf{N} : \mathtt{(evenp\ x)} \wedge \mathtt{x} < 17\}|$? •

Infinite sets have cardinalities too, but those cardinalities are not natural numbers and we do not discuss them here.

A

A Potted History of Logic in CS

In other sciences and engineering, the predominant mathematical systems are either statistics or differential equations. Statistics is most often used to describe systems whose behavior are subject to a lot of randomness or are so complex that we cannot describe them deterministically, e.g., voter behavior, the treatment outcomes for experimental cancer drugs, and quantum events. Differential equations are most often used with systems whose behavior is mechanical and continuous, e.g., physical motion, forces, wave behavior, etc.

But those mathematical systems are often unsuitable for computer science. The artifacts we study are digital. They behave exactly as we have programmed them to behave – following utterly precise mechanical rules. But they are discrete, not continuous. In one state a variable is `23` and in the next it is `17` with no intermediate state where it is `19.5`. Symbolic logic is perfect for reasoning about discrete systems that transition from state to state following fixed rules.

But there is another reason symbolic logic is so critical to computing. Computing finds applications everywhere, from games to new business paradigms, from medical diagnosis and rural health care delivery to nuclear weapons, from personal music players to climate modeling. How can one mathematical system be "predominant" across such a wide range of applications?

The answer is because symbolic logic is mankind's attempt to explain mathematically how we think. If some precisely defined concept has some property and you can rationally explain why it has that property, then you can use symbolic logic to prove it.

But if you can explain why the property holds, why use mathematical logic? It turns out that very often our "rational explanations" are only partly accurate! Often we forget about special cases or just slip up. Those kinds of bugs in our thinking very often turn into bugs in our programs or our designs. Symbolic logic lets us write things down with formulas and manipulate the formulas – it let's us put our thoughts in writing and gives us rules for going from one conclusion to the next.

In some deep sense, anything that is really precise can be said in logic. So the language is much richer than anything you've seen in algebra, trig, or calculus. Logic also gives you a fixed set of rules you can use to transform one formula into another – and these rules come with an amazing warranty: if you reduce one formula to another with these rules and only use valid axioms, then every deduction you make will be valid.

The origins of logic go back thousands of years to the ancient Greeks. Socrates, Euclid.

and Aristotle all played important roles in codifying how we think. Almost 2000 years after them, Leibnitz (17th century) tried to reduce all of human thought to a kind of calculus with which he hoped to be able to derive fundamental logical and even ethical truths. He even imagined a computing machine that would implement his logic. In the 19th century Boole, Frege, Peano, and Russell began to lay the foundations of modern logic, formalizing in a nearly algebraic way the logic of propositions ("if p and q, then r"), quantifiers ("for all x, there is a y such that ..."), integers and induction (how to prove something is true for all the integers), and the explorations of the power of the formalisms. Finally, in the 20th century, Hilbert, Herbrand, Godel, Church, Turing, and von Neumann developed logic into its modern form as the theoretical foundation of all mathematical thinking.

In the course of these many centuries of development, the goals of research into mathematical logic changed. One reason for the change was that the calculus of Newton and Leibnitz gave physical scientists a powerful mathematical tool to explain the natural world. This decreased the need for a logical calculus. Another reason was that in the 19th and 20th centuries, logicians began to accept that trying to discover ethical or emotional "truths" by mathematical reasoning was farfetched – at best a radical simplification of how the human mind works. People are not rational.

But by then they had also realized that the very questions "what is truth?" and "how can we know something is true?" were very deep and could be explored with mathematical reasoning by studying formal logical systems. Thus, the goals morphed from providing a systematic way to discover truth into models of how truth is established. Logicians tailored logic so that logic is "easy" to study and powerful enough (theoretically) to prove anything that can be proved. Logicians then investigated the expressive limits of such logical systems. This led to some of the most profound discoveries in our history, the most famous being that of Gödel who showed there are mathematical truths that cannot be proved and Turing who showed there are computational questions that cannot be answered by computer programs.

A side-effect of this transformation of the goals of logic was that logic – as studied by logicians – is practically impossible to actually use to establish truth! This affected how logic was taught and how textbooks about it were often written.

The history of science is full of seemingly amazing coincidences and one of them is the connection between logic and computation. The goals of logic shifted in part because the natural world didn't need it – calculus was the mathematics to explain falling objects, flying objects, steam engines, space travel, and virtually every machine men and women could build. But the very people who explored the limitations of logic as a means of finding truth, including Turing and von Neumann, were simultaneously creating a new kind of machine – a machine whose operation could could only be described and understood by using mathematical logic.

Computer hardware and software operate according to fixed rules. The very aspects of logic that made it unsuitable for modeling ethical truths and falling objects – its insistence that something is either black or white, true or false, with no imprecision or vague half-truths – make it perfect for modeling computation. And as our machines and software increased in complexity, logic became virtually the only tool available to analyze how they behave.

These observations make mathematical logic the mathematics for computer science. If you want to understand computer science, you have to understand logic in the same way that if you want to understand physics you must understand calculus.

B

On Congruences

An equivalence relation (see page 265) is any relation that is reflexive, symmetric, and transitive. The most basic equivalence relation is equality, formalized by `equal` or "=". But there are many others.

`Iff` is an equivalence relation. It is reflexive, symmetric, and transitive. To use an equivalence other than equality to replace "equivalents by equivalents" one needs "congruence lemmas". To illustrate them, consider `iff`.

If the term α at address π admits propositional replacement in γ as defined on page 97, and γ' is the result of replacing that occurrence of α in γ by a propositional equivalent β, then it is possible to derive the propositional equivalence or sometimes even the equality of γ and γ' by chaining together the following *congruence* lemmas.

```
 1.  (implies (iff a b) (equal (if a x y) (if b x y)))
 2.  (implies (iff a b) (iff (if x a y) (if x b y)))
 3.  (implies (iff a b) (iff (if x y a) (if x y b)))

 4.  (implies (iff a b) (equal (not a) (not b)))

 5.  (implies (iff a b) (equal (and a p) (and b p)))
 6.  (implies (iff a b) (iff (and p a) (and p b)))

 7.  (implies (iff a b) (iff (or a p) (or b p)))
 8.  (implies (iff a b) (iff (or p a) (or p b)))

 9.  (implies (iff a b) (equal (implies a p) (implies b p)))
10.  (implies (iff a b) (equal (implies p a) (implies p b)))

11.  (implies (iff a b) (equal (iff a p) (iff b p)))
12.  (implies (iff a b) (equal (iff p a) (iff p b)))
```

All these congruences have the hypothesis `(iff a b)` and all then tell us that we can replace `a` by `b` in certain positions of certain functions and preserve either equality (congruences 1, 4, 5, and 9–12) or just propositional equivalence (congruences 2, 3, and 6–8). By chaining these rules together and using the reflexivity, symmetry, and transitivity of `iff` you can justify replacement of propositional equivalents by propositional equivalents at any address admitting propositional replacement, no matter how deeply it occurs in the larger term.

In general, a *congruence* lemma is of the form:

$(eqv_1$ `a b`$) \rightarrow (eqv_2$ $(f$ `...` `a` `...`$)$ $(f$ `...` `b` `...`$))$,

where eqv_1 and eqv_2 are equivalence relations. It tells us that two calls of f return "equivalent" results (in the eqv_2 sense of "equivalence") if one replaces a given argument, `a`, by any "equivalent" value b (in the eqv_1 sense of "equivalence").

Often, eqv_1 and eqv_2 are the same, as they are in congruence 2 above. But sometimes they are different, as in congruence 1.

Once you have proved that a relation is an equivalence relation and you have proved the congruence theorems stating that that sense of equivalence holds between two terms, you can use the relation just like you would an equality – provided you confine yourself to addresses that "admit" that equivalence. The congruence lemmas tell you where those addresses are.

Another oft-used equivalence relation is the notion of one list being a permutation of another, formalized by our `perm` (page 78). But let's write "`(perm x y)`" in a way that suggests it is a kind of "equivalence," say "`x` ≃ `y`". First, "≃" is an equivalence relation: it is reflexive, symmetric, and transitive.

Now suppose we're trying to simplify a term like:

```
(mem e (rev (isort d)))
```

to something that is propositionally equivalent. Suppose we've proved these facts about `perm` (aka "≃")

```
perm-mem-congruence:
(a ≃ b) → ((mem e a) ↔ (mem e b))

perm-rev-congruence:
(a≃b) → ((rev a)≃(rev b))

perm-isort:
(isort x) ≃ x

perm-rev:
(rev x) ≃ x
```

The first congruence lemma, `perm-mem-congruence`, tells us that while simplifying `(mem e (rev (isort d)))` to maintain propositional equivalence we can rewrite its second argument maintaining ≃. So when we rewrite the interior `(rev (isort d))` we will maintain ≃. But the second congruence lemma tells us that if we rewrite a `(rev (isort d))` maintaining ≃, we can replace its argument by something that is ≃. So when we rewrite the interior `(isort d)` maintaining ≃, we can use the third lemma, `perm-isort`, as a replacement rule: `(isort d)` ≃ `d`. So we rewrite `(rev (isort d))` to `(rev d)`, knowing they are ≃. But the fourth lemma, `perm-rev`, says we can rewrite that (maintaining ≃) to `d`. In summary, exploiting the transitivity of ≃ and the four rules above, we can simplify

```
(mem e (rev (isort d)))
```

to `(mem e d)` while maintaining propositional equivalence.

C

On Entailment

You will sometimes see people write logical formulas connected by the signs "$\vdash$" or "$\models$". We briefly discuss this notation.

The notation

$$\vdash \beta$$

means that β can be proved in whatever logical system or theory the author is talking about. We have been writing this

Theorem: β.

More generally,

$$(\alpha_1, \ldots, \alpha_k) \vdash \beta \tag{C.1}$$

means that β can be proved from the axioms α_i.

Sometimes it is more convenient to write it "backwards" as in

$$\beta \dashv (\alpha_1, \ldots, \alpha_k) \tag{C.2}$$

as when we "reduce" the proof of β to a proof of α and write "$\beta \dashv \alpha$".

Sometimes the author is dealing with some particular logical theory containing some previously given axioms $\mathcal{A}$ and in such a setting (C.1) really means

$$(\mathcal{A}, \alpha_1, \ldots, \alpha_k) \vdash \beta$$

That is, β can be proved from the "usual axioms" $\mathcal{A}$ *plus* the α_i. Some authors will write this as

$$(\alpha_1, \ldots, \alpha_k) \vdash_{\mathcal{A}} \beta$$

Most of the time, authors are only dealing with one theory which is understood and the $\mathcal{A}$ is just omitted entirely. That is what we'll do henceforth.

If the α_i are closed formulas then (C.1) is equivalent to

$$\vdash (\alpha_1 \wedge \ldots \wedge \alpha_k) \rightarrow \beta. \tag{C.3}$$

The equivalence of (C.1) and (C.3) for closed α_i means that if you want to prove $(\alpha_1 \wedge \ldots \wedge \alpha_k) \rightarrow \beta$ you can assume the α_i as though they were axioms and then derive β. You can move *closed* α_i back and forth across the "$\vdash$" sign, making them axioms on the left or hypotheses on the right. But this is so *only* if the α_i in your goal theorem are closed.

Because ACL2 lacks explicit quantifiers, the α_i in our conjectures are never closed (unless they have no variables at all). But you might someday find yourself trying to prove a formula like

$$((\forall x, y, z : \alpha_1) \wedge \alpha_2) \rightarrow \beta$$

where the free variables of α_1 are x, y, and z. In this case, you can just assume $(\forall x, y, z : \alpha_1)$ as an axiom and try to prove

$$\alpha_2 \rightarrow \beta$$

If α_1 is an ACL2 formula, then

(defaxiom *name* α_1)

has the same effect as assuming the axiom $(\forall x, y, z : \alpha_1)$. That is, while proving $\alpha_2 \rightarrow \beta$ you are free to use arbitrary instances of α_1.

You will also sometimes see people write:

$$(\alpha_1, \ldots, \alpha_k) \models \beta \tag{C.4}$$

This is different. It means that any model that makes all of the α_i true makes β true. Again, you can move *closed* α_i back and forth across the "$\models$" sign, making them hypotheses on the right.

In a *complete* logical theory, everything that is valid can be proved, so "$\models$" and "$\vdash$" are equivalent.

But Gödel showed that sufficiently rich theories are necessarily incomplete: not every truth can be proved. ACL2 is such a theory. Indeed, any first-order theory of the natural numbers (including induction) with addition and multiplication is incomplete!

So there are valid formulas that cannot be proved. So knowing that β is true in all the models satisfying the α_i (as (C.4) says) does not necessarily mean that you can prove β from the α_i (as (C.1) says).

D

ASCII Substitutes

If you type your homework I recommend using the substitutions shown below.

Follow the guidelines for pretty printing on page 31.

Spell out Greek letters. Do not use any of the names of Greek letters or the special infix symbols below as variable symbols. For example, do not use "alpha", "in", or "cross" as variable symbols.

Below are all the ASCII substitutions mentioned in the text.

<table>
<tr><td>$x \Longrightarrow c$</td><td>x ==> c</td><td>evaluates to</td></tr>
<tr><td>$x = y$</td><td>x = y</td><td></td></tr>
<tr><td>$x \neq y$</td><td>x != y</td><td></td></tr>
<tr><td>$(p \wedge q)$</td><td>(p && q)</td><td></td></tr>
<tr><td>$(p \vee q)$</td><td>(p || q)</td><td></td></tr>
<tr><td>$(\neg p)$</td><td>(~ p)</td><td></td></tr>
<tr><td>$(p \rightarrow q)$</td><td>(p --> q)</td><td></td></tr>
<tr><td>$(p \leftrightarrow q)$</td><td>(p <--> q)</td><td></td></tr>
<tr><td>$(x \times y)$</td><td>(x * y)</td><td>arithmetic multiplication</td></tr>
<tr><td>$(x \times y)$</td><td>(x cross y)</td><td>cross-product</td></tr>
<tr><td>$(x \leq y)$</td><td>(x <= y)</td><td></td></tr>
<tr><td>$(x \geq y)$</td><td>(x >= y)</td><td></td></tr>
<tr><td>$v \leftarrow y$</td><td>v <-- y</td><td>substitution binding</td></tr>
</table>

$(\forall v : \psi)$	`(all` v ψ`)`	
$(\forall v_1, v_2 \ldots : \psi)$	`(all (`v_1 v_2 `...)` ψ`)`	
$(\forall v_1, v_2 \ldots \in S : \psi)$	`(all (`v_1 v_2 `...) in` S ψ`)`	
$(\exists v : \psi)$	`(exists` v ψ`)`	
$(\exists v_1, v_2 \ldots : \psi)$	`(exists (`v_1 v_2 `...)` ψ`)`	
$(\exists v_1, v_2 \ldots \in S : \psi)$	`(exists (`v_1 v_2 `...) in` S ψ`)`	
$\psi \dashv \phi$	ψ `--!` ϕ	
$(p \leftarrow q)$	`(`p `<--` q`)`	reverse implication
$\{x : \psi\}$	`(setof` x ψ`)`	
$\{(f\ v_1\ v_2\ \ldots) : \psi\}$	`(setof` $(f\ v_1\ v_2\ \ldots)$ ψ`)`	
$\{e_1, e_2, \ldots\}$	`(set` e_1 e_2 `...)`	
$(e \in S)$	`(`e `in` S`)`	
$(e \notin S)$	`(`e `!in` S`)`	
$(R \subseteq S)$	`(`R `subeq` S`)`	
$(R \cup S)$	`(`R `un` S`)`	
$(R \cap S)$	`(`R `int` S`)`	
$(R \setminus S)$	`(`R `minus` S`)`	
$(\wp\, S)$	`(pwr` S`)`	power set
$(\bigcup S)$	`(big-un S)`	
$(\bigcap S)$	`(big-int S)`	

Bibliography

1. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, Boston, MA., 2000.
2. J. R. Shoenfield. *Mathematical Logic.* Addison-Wesley, Reading, Ma., 1967.
3. R. R. Stoll. *Set Theory and Logic.* Dover Publications, New York, 1963.
4. A. N. Whitehead and B. Russell. *Principia Mathematica (Vol 1–3).* Cambridge University Press, 1910. (See also http://name.umdl.umich.edu/AAT3201.0001.001).

Index

Underlined words are the names of links in the online documentation. From the ACL2 home page, `http://www.cs.utexas.edu/users/moore/acl2`, select the link to the User's Manual and then the link to the Index of all documented topics.

⇒ (logical implication), 118
≡ (logical equivalence), 118
∃, 196
∀, 191
∧, 118
↔, 118
→, 118
¬, 118
∨, 118
∼ (logical negation), 118
⊃ (logical implication), 118
*, 48
+, 48
-, 48
/, 48
<, 48
¡=, 56
= (equality), 118
¿, 56
¿=, 56
! (logical negation), 118
* (logical and), 118
--> (logical implication), 118
- (logical negation), 118
<--> (logical equivalence), 118
==> (logical implication), 118
&& (logical and), 118
& (logical and), 118
|| (logical or), 118
| (logical or), 118
1:1, 272

ABORTING from raw Lisp, 33, 199
abstract syntax, 117
accumulator, 63
actual expressions, 10
actual values, 10
actuals, 10
add-to-each, 78
addition, 48
address (of an occurrence), 98
admits propositional replacement, 99
all-all, 234
all-and, 231
all-and-1, 231
all-and-2, 231
all-not, 233
all-or-2, 233
all-partitions, 268
alternative-true-listp, 176
and, 49
antecedent, 51
antisymmetric, 263
app, 73
app-rev-nil, 181
app-right-id, 178
application, 6
apply, a substutition, 109
arity, 6
ASCII substitutes, 285
assoc-of-app, 177
asymmetric, 263
atom, 34

base case, 170
binary relation on set, 262
binding occurrence, 199, 246
binding, in a substitution, 109
body (of defined function), 10
body (of quantified expression), 192

body (of set builder expression), 246
boolean-ids, 106
booleanp, 49
bound occurrence, 199, 246
bound variable, 199
bound variable list, 200
bound variable of quantifier, 192

call, 6
capture, 201
case sensitive, 14
Cases Rule, 129
casting, 57
cell, of partition, 267
closed formula, 199
closure, universal, 199
complete logical system, 284
composition, 272
Computation (Comp) Rule, 130
conclusion, 51
concrete syntax, 117
conditional rewrite, 129
congruence, 279, 280
conjoining (a list of terms), 111
conjunct, 50
conjunction, 50
conjunction (of a list of terms), 111
conjuncts list, 111
connected, 263
connectors, propositional, 49
cons, 48
consequent, 51
consp, 48
Constant (Const) Rule, 130
constant function, 59
contrapositive (of an implication), 52
converse (of an implication), 52
converse of relation, 262
copying the hypothesis, 205
corollary, 128
counterexample, 157
countermodel, 158
cross product, 259

del*, 73
del1, 72
destination of edge, 236
diff, 76
difference, of lists, 76
disjoint sets, 267
disjunct, 50
disjunction, 50
disjuncts list, 112
division, 48
domain, 77
domain of relation, 262
domain, of a substitution, 109
domain, of table, 77
doubletp, 52

edge of graph, 236
entailment, 283
environment, 8
equal, 48
equivalence class, 265
equivalence classes induced, 269
equivalence of factored formula, 115
equivalence relation, 265
equivalence, propositional, 52
equivalent, propositionally, 97
exclusive or, 52
exist-and-1, 232
existential quantifier, 196
exists-all-1, 235
exists-exists, 235
exists-not, 233
exists-or, 233
exists-or-1, 233
exists-or-2, 233
expression, 13

factor, 113
false branch, 11
findpos, 78
finite directed graph, 236
finite path, 237
first, 48
fn-q37, 26
fn-q31, 25
fn-q33, 26
fn-q35, 26
fn-q36, 26
formals, 10
formula, 13
formula, propositional, 49
free occurrence, 199
free variable, 199
function, 3, 6, 271

function symbol, 6
functional programming, 4
functions, propositional, 49

hit, by substitution, 109
`how-many`, 74
hypothesis, 51
Hypothesis (for Quantified Formulas) (qHyp) Rule, 211
Hypothesis (Hyp) Rule, 129

identical, 97
identity, 93, 271
`if`, 48
`if-distrib`, 159
`if-if`, 159
`if-split`, 159
`iff`, 49
`image`, 78
image of function on set, 272
image, via table, 77
implication, 51
`implies`, 49
`implies-all`, 234
in the scope of a quantifier, 200
`in-tablep`, 77
indistinguishable, propositionally, 52
inducing a partition, 269
induction step, 170
`insert`, 80
instantiate, 109
instantiating, 15
instantiating qwff, 210
`int`, 75
`integerp`, 48
intersection, of lists, 75
invalid, 158
inverse, 274
invocation, 6
irreflexive, 263
`isort`, 80
`isort-id`, 182
`isort-isort`, 182

justification, 9

lemma, 128
`len1`, 59
less than relation, 48
linear order, 264
Lisp, 3
`listcopy-id`, 176
logical negation, 51
`lookup`, 77

`make-table`, 78
map, 77
`mapnil-app`, 179
`mapnil-rev`, 179
matrix, of quantified expression, 192
meaning, 192
`mem`, 65
`mem-app`, 179
mini-steps of rewrite, 85, 102
model, 157
multiplication, 48
`mutual-recursion`, 58

`natp`, 48
natural number, 10
negation, 51, 112
negation, pointwise, 112
`nfix`, 57
`no-dups`, 74
node of graph, 236
non-strict simple order, 264
`not`, 49
`not-all`, 233
`not-exists`, 233
`nth`, 70

one-to-one, 272
onto, 272
operators, logical, 49
`or`, 49
`ordp`, 79
`ordp-insert`, 181
`ordp-isort`, 181

`pair`, 256
`pairp`, 256
parameters, formal, 10
partial order on, 264
partition, 267
pattern of factored formula, 115
`perm`, 78
`perm-app1`, 183
`perm-app2`, 184
`perm-cons`, 183
`perm-isort`, 184

perm-reflexive, 182
perm-rev, 184
perm-rev-id, 184
perm-symmetric, 182
perm-transitive, 183
pointwise negation, 112
predicate, 13
prefix syntax, 6
pretty printing, 31, 119
proof, 127
propositional equivalence, 49, 52, 97
propositional formula, 49
propositional functions, 49
propositional simplification, 154
pure ACL2 objects, 244

quantifiers, 191
quod erat demonstrandum, 144

range, 77
range of relation, 262
range, of a substitution, 109
recognizer, 12
recursive, 16
refactoring, 115
reflexive, 263
relation, 262
relieving hypotheses, 128
relieving the hypotheses, 128
rem-dups, 74
replacement of factored formula, 115
replacing (an occurrence), 98
rest, 48
rest-nest around, 170
rest-nest substitution, 170
rev, 74
rev-app, 179
rev-rev, 179
rev1-rev, 181
rev1-rev-general, 180
reversed turnstyle, 198
Rewrite (for Quantified Formulas) (qRe) Rule, 210
Rewrite (Re) Rule, 128
right-associate, 80
roster notation for set, 246
rule of inference, 127

same-elements, 75
sentence, 199
set builder notation, 245
set comprehension, 245
set theory, 243
signature, 272
simple path, 238
singleton list, 52
source of edge, 236
spine, 37, 38
stringp, 48
strong simple order, 265
strongly connected, 263
structural induction, 169
subeq, 75
subset relation on lists, 75
substitution, 109
subtraction, 48
symbolp, 48
symmetric, 263

table, n-column, 76
tablep, 76
tail-recursive, 180
Taut Rule, 128
tautologies, 131
tautology, 106, 107
tautp, 107
term, 6
test, of if, 11
textlikep, 53
theorem, 127
Thm Rule, 128
total, 263
total order, 264
transitive, 263
true branch, 11
true list, 40, 64
true-listp, 40, 64
true-listp-rev, 181
truth table, 132
truth values, 123
Turing complete, 30
turnstyle, 209
typed language, 49

uaf, 27
union, of lists, 75
universal closure, 199
universal quantifier, 191

universe, 192
`unn`, 75
untyped language, 48
used propositionally, 97

valid, 83
vertex of graph, 236

weak simple order, 264

`xor`, 52

zp, 56