The CS340d Manual

Version (month.week.day): 4.3.21
Updated 22 April 2023, early afternoon

Continue reading the Recursion and Induction notes through topic **r-and-i-definitions-revisited**.
You may start by jumping to "**recursion-and-induction ...**" (`http://acl2.org/manual`)

Suggested viewing:
2021 Summer School Video (J Moore), at
`https://youtu.be/pVRfeu8MbgE`

See new Lab: **Complete**

See new homework: **Complete**

Warren A. Hunt, Jr. (`hunt@cs.utexas.edu`)

Texinfo version of the documentation for UTCS CS340d; originally by Warren A. Hunt, Jr.

Short Contents

Table of Contents

1 Introduction

This course, Debugging and Verifying Programs (CS340d), introduces students to rigorous (sometimes formal) specification and (analytic) analysis techniques that should help them be better programmers. We will introduce, analyze and apply tools for confirming program correctness by proof methods. Some of our methods will be practical — rules of thumb, or just suggestions for successful coding. Other methods will involve using mathematics to write specifications; and subsequently, performing proofs to assure that code is meeting its specification.

This document template will be populated throughout the course and contains information to help students navigate the UT CS340d course, Debugging and Verifying Programs. This information is arranged in a hierarchical manner and will be updated as the semester proceeds. We would appreciate receiving suggestions for improvements in all aspects of our course, including this information, classroom activities, presentations, assignments, laboratories, and anything else related to this class. Thus, comments, criticisms, assistance, ideas, examples, and improvements are welcome.

1.1 Course Announcement

In this class, we will consider how sequential programs are specified and how the correctness of their implementations are confirmed. This class will require careful thought as we will be pushing the boundaries of what the academic community considers to be an adequate specification and sufficient confirmation evidence that a program meets its specification. Typically, some form of testing is the only mechanism that is used to see if a program meets its specification – this class will investigate both testing and other verification methods.

To develop skill in program specification, analysis, verification, and debugging, we will assign a litany of problems where students will be expected to write specifications, write code that meets these specifications, produce arguments that defend their claim that their solutions meet the specifications, and write reports about their efforts.

Note that this course carries a writing flag, and students will write more often than is typical in other CS courses. Students will also be asked to address problems where they will need to decide whether various implementations produced by others are correct, and to debug these programs when they are not correct.

To be able to debug programs, we will investigate common debugging and analysis tools. To be able to use such tools effectively, it will be necessary for students to understand how binary is used to direct a processor. Students will need to understand how binary programs are organized and also how to inspect such binary programs during their execution.

Another component of debugging and verification is a well-organized method for program development. Version control for code and documentation is widely used and is generally necessary – especially in multi-person teams. It can also be necessary and effective tool for a single person dealing with a complex project structure or very large numbers of software components in a system.

In many cases, we will use proof-based techniques to determine the correctness of our code. At first, we will investigate hand proofs; that is, we will use some informal notation to compare a specification program to an implementation program. We will also convert the

behavior of some programs into a form that will allow a mechanical comparison of the behavior of two programs.

This class will be taught in an "inverted" style. That is, we will concentrate class time on examples, working through code, describing challenges, and exploring problems being faced by students working on homework sets or larger laboratory projects. Thus, it is important that you bring your laptop to everyclass. There will be lectures to introduce various topics, but primarily, we will use class time for problem solving, demonstrating how to use various tools, and exchanging information.

Most of the information needed for this course will be provided; however, we do expect students to have internalized information from their algorithms and data structure courses. In addition, we will make use of the material in "Computer Systems, A Programmer's Perspective", 3rd Edition (the CS429 textbook). In our use of the Y86 ISA, we may occasionally refer to Intel's specification for the X86. Information not provided we be readily available on web, such as the programming information available from Agner Fog `http://www.agner.org/optimize/` website. And, students may wish to have occasional access to "Hacker's Delight, 2nd Edition" by Henry S. Warren, Jr.

In addition to being a UTCS undergraduate student, the prerequisite for CS340D is the successful completion of CS429. There is no textbook required for CS340D, but we will sometimes refer to your CS429 textbook ("Computer Systems, A Programmer's Perspective" Third Edition, by Randal E. Bryant and David O'Hallaron, Prentice Hall). In addition, we will sometimes make use of the second edition of the book "Hacker's Delight" by Henry S. Warren, Jr. This book will be used for various background problems, and some of the homework and programming assignments may be based on the material from this book. We will make use of other web-based material as needed and references will be provided in class. There is a website (`https://github.com/lancetw/ebook-1/blob/master/02_algorithm/Hacker%27s%20Delight%202nd%20Edition.pdf`) associated with the "Hacker's Delight" book. And another great source of problems is the Hakmem website (`http://www.inwap.com/pdp10/hbaker/hakmem/hakmem.html`).

Tests and quizzes are open-book, open-notes affairs – however, no electronic devices (laptops, cell phones, tablets, PDAs, calculators, etc.) of any kind are allowed during test and quiz events. As such, you may wish to have a physical copy of any materials that you believe will be helpful. Remember, cell phones are not allowed during exams.

This course will require students to write programs in Lisp, and occasionally C. Knowledge of assembler will also be critical as it is the binary code that really determines what programs do – and it is during binary execution on physical hardware that code will fail. It is recommended that you have access to "The C Programming Language" (`https://en.wikipedia.org/wiki/The_C_Programming_Language`), Second Edition, by Brian Kernighan and Dennis Ritchie, Prentice Hall Software Series. For examples and help with C-language use, you will find that there are many Web pages devoted to C-language programming.

Why do we use C, or its extension, C++? C is the language that is used to implement many systems, such as FreeBSD, Linux, MacOS, Windows, as well as many user tools (e.g., wc, grep, ed, sed, emacs,...). Java programmers should have no problem with the subset of C that we will use, but Java programs are not generally used to interface with assembly language programs. Students might be introduced to processor-specific-language capabilities, such as referencing x86 processor-specific counters, that lie outside of the official C-language

definition. Students will be introduced briefly to the GNU Debugger ("gdb") program; we will use a tiny subset of "gdb" as a model for one of the class laboratories.

While C will be used intermittently throughout this course, most of our programming will be done in ACL2. ACL2 is a subset of Common Lisp that can be used to develop models of digital systems and prove properties of these models using a theorem prover included with the system. Depending on time, and the skills and interests of the students in this class, students may wish to develop a BDD (binary decision diagram) package. We will also investigate and use a SAT solver. For such analysis tools, we will use the Z3 and ACL2 systems, and work on several examples of using logic to verify hardware circuits or assembler program models.

In some cases, it may be helpful to reference documentation about the x86 architecture. In some cases, students may be asked to read small sections of x86 documentation. Note that these documents are large – these documents are indicative of the complexity of the x86 architecture and, in general, of modern computer systems. Companies other than Intel (AMD and VIA) that develop and market x86 processors must fully comprehend the information contained in these documents. Unfortunately, the information contained in these documents is insufficient to develop a competitive x86 processor implementation. AMD offers their own manuals, which can be a place to look if the Intel documentation isn't sufficient. VIA doesn't publish any specification for their X86 implementations. Note, there are many undocumented features (e.g., caching read-ahead strategies, I/O ordering behavior, virtualization, context-switch mechanisms, encryption-and-decryption instructions, etc.) that are necessary to make an x86 processor perform well on a litany of common benchmarks. For an x86 processor to be competitive, it will need to contain some of these features. Although many x86 implementation mechanisms are protected by patents, anyone is now free to build their own 32-bit x86 implementation using Intel's IP as all of Intel's patents specific to the (32-bit x86) Pentium have expired.

For the adventurous student, special projects are possible. The content of a special project is pretty flexible – so long as it has to do with specification and validation. For instance, we are interested in the development of an ISA (instruction set architecture) model of IBM's Harvest computer, which was a extension of IBM's Stretch computer. Another possible specification project might involve some older microprocessor, e.g., the Motorola 68030 or the National Semiconductor NS32032. Or, a student might wish to formally specify RISC-V. Another project of high interest concerns booting FreeBSD or Linux on our evolving ACL2-based x86 ISA emulator. Other independent study projects are possible; please discuss your particular interest in one of the above projects or some new ideas with the instructor.

The value you get from this class will be directly related to the effort you (as a student) put forward. This class will require that you learn to work on your own. You may find this class to be less structured than many of the classes you have previously taken. For instance, from experience in teaching CS429 many times, we know that lecture time dominates the CS429 class. In this class, there will be one or two short (less than 15 minute) lectures, but not nearly as many nor as long as is typical in CS429. The majority class time will be used to directly address problems and seek their solutions. Students will need to have access to a computer during class.

In class, we will be doing some real-time programming to support discussing various issues, such as how the debugger or version control system functions. Eventually, all programming assignment must work on the CS Department Linux machines, but being able to use your

own IDE (Integrated Development Environment) may speed up your work and you may end up with more tools on your programming toolbox than when you started this class. When we are discussing programming issues and working together on coding it may be helpful for you to try things immediately. Note, if needed, it is possible to checkout a Linux-based laptop from the UTCS Department. You can check with the instructor if you wish to borrow such a laptop.

Students will be encouraged to give short (five- to ten-minute) presentations in class on particular topics. When well done, these presentations can serve in place of a missed quiz or homework. In fact, any student may be called upon to give a two- or three-minute presentation on something being discussed in class or on their solution to a homework problem. Please come to class prepared to work.

We will sometimes stop our classroom activities for a few minutes to give everyone a chance to consolidate their thinking. During this time you might formulate questions that can help you and your fellow students overcome problems of general concept understanding or with questions about the in-class presentations.

Our office hours are listed on the main class web-page. In addition, if you need help, you may certainly seek out and visit with the class TA and/or the instructor(s). You may arrange to meet us at times other than those listed, but you will need to send E-mail to arrange a time. If we become too busy during the scheduled office hours, we will expand our office hours to meet the needs of the students. If you cannot come to the scheduled office hours due to conflicts with other classes, let us know quickly so we can make arrangements to meet your needs.

1.2 Class Syllabus

The following gives an outline of the topics we will cover in this course. We are open to discussing other topics of general interest, and we will include some of our own experience in hardware and software verification.

The syllabus below is approximate; the exact rate at which we will cover some of the material is hard to predict. So the schedule may vary. All changes to this schedule will be announced in class and broadcast to the course CANVAS page. Additional summary information about the class laboratory and homework assignments will be made available as the course progresses.

```
  Schedule Below is Approximate, Lectures Dates/Topics May Change Slightly

  *** NOTE:  Bring Laptops to every class; we will access ACL2 on
  ***        UTCS Linux machines during class.

  *** NOTE:  Quizzes can and will occur during nearly every class period.
  *** NOTE:  At-Your-Desk Problems will be pursued during class.

  *** NOTE:  Due dates for Homework and Labs are tentative until assigned.


Week  Class    Date   Short Description
```

```
0     00     Jan 10  Course Content Introduction,
                     Course Procedures and UT required disclosures
                     Lisp/ACL2 Introduction, Fibonacci function

0     01     Jan 12  Writing CS314/CS331 algorithms in Lisp.  The
                     simplest kind of verification -- co-simulation.


1     02     Jan 17  Introduce DEFUN and TRACE$, use TRACE$ to investigate
                     control-flow during execution of a function.

1     03     Jan 19  More on using TRACE$ with ACL2 functions; LEN, APP,
                     FACT, TREE-COPY, MERGE-LISTS


2     04     Jan 24  Lookup and update.  Duplicate detection.  Set intersection,
                     union, and negation operations.

* * * * *    Jan 25  Last Day to drop class without permission

2     05     Jan 26  Terms and evaluation.  Mutual recursion.


3     06     Jan 31  Introduction to ACL2 ``guards''; data and
                     structure recognizers; well-formed inputs; trees

3     07     Feb  2  General correctness principles, assertions,
                     invariants


4     08     Feb  7  Presentation and use of the ACL2 logic

4     09     Feb  9  Basic ACL2 data types


5     10     Feb 14  ACL2 Terms

5     11     Feb 16  Substitution


6     12     Feb 21  Function definitions,
                     Model a memory using NTH and !NTH

6     13     Feb 23  ACL2 Axioms


7     14     Feb 28  Sorting with our memory model
```

```
 7     15     Mar  2  Axioms, basic hand proofs

 8     16     Mar  7  Terms

 8     17     Mar  9  Structural Induction

        Mar 12-18  Spring Break

 9     18     Mar 21  Practice with list recursion

 9     19     Mar 23  Practice with list-based memory modeling

10     20     Mar 28  Definitional Principle

10     21     Mar 30  Induction Principle

11     22     Apr  4  Relationship between recursion and induction

11     23     Apr  6  Tree-based algorithms

12     24     Apr 11  ACL2-based verification

12     25     Apr 13  Machine modeling

13     26     Apr 18  Function profiling

13     27     Apr 20  Construction of the various data types

14     28     Apr 25  Example uses of the ACL2 system

14     29     Apr 27  Two quizzes, stump the professor
```

1.3 Writing Flag

As a future employee, one of the most important things you will need to do is to be able to communicate with your co-workers and your customers. Yes, you may be a programmer, and your work output may be code, but you will need to write descriptions of what your

code does, write reports to document your efforts, write documentation for your code, or write proposals for funding new projects.

As this course includes the writing-flag designation, students will be asked to write more often than is typical in other CS courses. Students will be asked to address problems where they will need to decide whether various implementations are correct, to debug the programs when they are not correct, and to articulate clearly what they have done in a technical report.

1.4 Homework

There will be ten to twelve homework assignments given during the semester. On most weeks, homework will be assigned on Thursdays and due seven (7) days later (on the following Thursday) by class time. No homework will be assigned the last two weeks of class, but there may be a homework due the last week of class. The two lowest homework grades will be dropped in the computation of the final homework grade.

Homework will not be accepted late! We repeat, no late homework!

1.5 Laboratory Projects

There will be four (0, 1, 2, and 3) Laboratory Projects assigned. Once a laboratory due date has arrived, material addressed in that laboratory may appear on a quiz or exam. Laboratory assignments are important; performing the work necessary to complete the class laboratories is the means by which you will solidify your understanding of the class material and the work that it takes to make you a better thinker and programmer.

Laboratory Projects may be turned in up to one week late, but no later than the last day of class. Late laboratory project submissions suffer a 20% reduction of the grade given for the content of the project. So a perfectly done laboratory assignment handed in late, can do no better than a maximum grade of 80%.

For each laboratory, a lab report will be part of the requirement. Remember, this course carries a writing flag, the quality and completeness of lab reports will count for 20% to 35% of the grade for the laboratory. So, it is important that you allot time and make a serious effort to provide the documentation required for each laboratory.

1.6 Quizzes

Over the course of the semester, there will be twenty, or more, in-class quizzes. Quizzes are ten- to twenty-minute affairs. In this course, no long exams will be given.

The material on quizzes will be cumulative, and we might even have two quizzes within a single class period. There will be no final exam.

1.7 Class Assessment

The weighting of the grades for the various aspects of the course are:

```
Component       Percentage of Course Grade
Quizzes:          40%  (any class period)
Homework:         30%  (submitted to an on-line system)
Labs:             30%  (see individual weighting just below)
```

The Laboratory Projects will be weighted as follows:

Laboratory	Percentage of Course Grade
Lab 0:	5%
Lab 1:	5%
Lab 2:	10%
Lab 3:	10%

The grading for the entire course will be as follows:

Course Score	Grade
[90 -- 100)	A
[87 -- 90)	A-
[85 -- 87)	B+
[80 -- 85)	B
[77 -- 80)	B-
[75 -- 77)	C+
[70 -- 75)	C
[67 -- 70)	C-
[65 -- 67)	D+
[60 -- 65)	D
[0 -- 60]	F

Note the interval marks around the course-score column. For example, a course grade of B will be assigned if your semester grade is greater than or equal to 80 and (strictly) less than 85. This also means that a course grade of at least 67 needs to be achieved for this course to count toward a UTCS degree – a grade of D+ or D is not considered a passing grade for a UTCS (student) major.

1.8 Class Advice

The students who do well in this class are survivors. This class is a fair amount of work, and it is important to keep current. The material in this class is cumulative, and it can be difficult to catch up if one falls behind. It is very important to keep turning in homework and laboratories. Generally, homework grades are our most reliable indicator of how well a student will do (or is doing) in this class. Note, it is very important to attend class, as quizzes will be given, and material that is not available readily may be discussed.

Due to the continuing uncertainty of the pandemic, we will be prepared to move our class to an online format – and this will require our use of Zoom. Any changes in how we will conduct class will be announced as the semester progresses.

1.9 Electronic Class Delivery

In the past few years, during the pandemic, we have offered this class electronically. This semester we will return to the more traditional "in-class" format. If you have any trouble with accessing class materials, submitting work, connecting to our class session, or any other issue that concerns your ability to function successfully, please do not hesitate to contact us. We will provide office hours multiple times each week so students may continue to engage with the Instructor and Teaching Assistant directly. In addition, some office hours will be offered electronically if there is sufficient student interest in having those available.

Students should be able to use Zoom, as we may hold extra sessions or even remote sessions by way of Zoom. Some office hours will also be held using Zoom; this allows us to schedule non-standard (e.g., evening) office hours when otherwise we will not be available.

1.10 Code of Conduct

The core values of the University of Texas at Austin are learning, discovery, freedom, leadership, individual opportunity, and responsibility. Each member of the University is expected to uphold these values through integrity, honesty, trust, fairness, and respect toward peers and community.

We believe that you belong here! Although UT is a very large organization, we are attempting to foster a climate conductive to learning and creating knowledge; we believe this is a basic tenant of people in our community. Bias, harassment and discrimination of any sort have no place here in our community. If you notice an incident that causes concern, please contact the Campus Climate Response Team (`http://diversity.utexas.edu/ccrt`).

In general, the information found in UT's Code of Conduct (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/Disclosures/CodeOfConduct.html`) is a good guide on how to conduct yourself in this class. Additional general information about College of Natural Sciences (CNS) class coursework and procedures can be found in former Vice Provost Laude's memorandum (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/Disclosures/CNS_Coursework_Routine_09-10.pdf`) to the CNS faculty.

This course attempts to comply with the requirements of the University and the State of Texas. Texas House Bill 2504 specifies a number of items regarding course materials and instructor qualifications (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/Disclosures/aug-2022.pdf`).

In addition, the material contained here and referenced are designed to be compliant with Gretchen Ritter's (Vice Provost for Undergraduate Education and Faculty Governance) August 3, 2012 memo (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/Disclosures/ritter-memo.txt`).

Ritter's memorandum also addresses issues concerning campus safety and security. Please familiarize yourself with this information, and let us know if you believe the class Website does not comply with any of these requirements.

```
                        Texas House Bill No. 2504

                                AN ACT

relating to requiring a public institution of higher education to
establish uniform standards for publishing cost of attendance
information, to conduct student course evaluations of faculty, and
to make certain information available on the Internet.

       BE IT ENACTED BY THE LEGISLATURE OF THE STATE OF TEXAS:


       SECTION 1.  Subchapter Z, Chapter 51, Education Code, is
```

amended by adding Section 51.974 to read as follows:

Sec. 51.974. INTERNET ACCESS TO COURSE INFORMATION.

(a) Each institution of higher education, other than a medical and dental unit, as defined by Section 61.003, shall make available to the public on the institution's Internet website the following information for each undergraduate classroom course offered for credit by the institution:
  (1) a syllabus that:
    (A) satisfies any standards adopted by the institution;
    (B) provides a brief description of each major course requirement, including each major assignment and examination;
    (C) lists any required or recommended reading; and
    (D) provides a general description of the subject matter of each lecture or discussion;
  (2) a curriculum vitae of each regular instructor that lists the instructor's:
    (A) postsecondary education;
    (B) teaching experience; and
    (C) significant professional publications; and
  (3) if available, a departmental budget report of the department under which the course is offered, from the most recent semester or other academic term during which the institution offered the course.

(a-1) A curriculum vitae made available on the institution's Internet website under Subsection (a) may not include any personal information, including the instructor's home address or home telephone number.

(b) The information required by Subsection (a) must be:
  (1) accessible from the institution's Internet website home page by use of not more than three links;
  (2) searchable by keywords and phrases; and
  (3) accessible to the public without requiring registration or use of a user name, a password, or another user identification.

(c) The institution shall make the information required by Subsection (a) available not later than the seventh day after the first day of classes for the semester or other academic term during which the course is offered. The institution shall continue to make the information available on the institution's Internet website until

at least the second anniversary of the date on which the institution initially posted the information.

(d) The institution shall update the information required by Subsection (a) as soon as practicable after the information changes.

(e) The governing body of the institution shall designate an administrator to be responsible for ensuring implementation of this section. The administrator may assign duties under this section to one or more administrative employees.

(f) Not later than January 1 of each odd-numbered year, each institution of higher education shall submit a written report regarding the institution's compliance with this section to the governor, the lieutenant governor, the speaker of the house of representatives, and the presiding officer of each legislative standing committee with primary jurisdiction over higher education.

(g) The Texas Higher Education Coordinating Board may adopt rules necessary to administer this section.

(h) Institutions of higher education included in this section shall conduct end-of-course student evaluations of faculty and develop a plan to make evaluations available on the institution's website.

SECTION 2. Subchapter E, Chapter 56, Education Code, is amended by adding Section 56.080 to read as follows:
Sec. 56.080. ONLINE LIST OF WORK-STUDY EMPLOYMENT OPPORTUNITIES. Each institution of higher education shall:

(1) establish and maintain an online list of work-study employment opportunities, sorted by department as appropriate, available to students on the institution's campus; and

(2) ensure that the list is easily accessible to the public through a clearly identifiable link that appears in a prominent place on the financial aid page of the institution's Internet website.

SECTION 3. Subchapter C, Chapter 61, Education Code, is

amended by adding Section 61.0777 to read as follows:
Sec. 61.0777. UNIFORM STANDARDS FOR PUBLICATION OF COST OF ATTENDANCE INFORMATION.

(a) The board shall prescribe uniform standards intended to ensure that information regarding the cost of attendance at institutions of higher education is available to the public in a manner that is consumer-friendly and readily understandable to prospective students and their families. In developing the standards, the board shall examine common and recommended practices regarding the publication of such information and shall solicit recommendations and comments from institutions of higher education and interested private or independent institutions of higher education.

(b) The uniform standards must:
  (1) address all of the elements that constitute the total cost of attendance, including tuition and fees, room and board costs, book and supply costs, transportation costs, and other personal expenses; and
  (2) prescribe model language to be used to describe each element of the cost of attendance.

(c) Each institution of higher education that offers an undergraduate degree or certificate program shall:
  (1) prominently display on the institution's Internet website in accordance with the uniform standards prescribed under this section information regarding the cost of attendance at the institution by a full-time entering first-year student; and
  (2) conform to the uniform standards in any electronic or printed materials intended to provide to prospective undergraduate students information regarding the cost of attendance at the institution.

(d) Each institution of higher education shall consider the uniform standards prescribed under this section when providing information to the public or to prospective students regarding the cost of attendance at the institution by nonresident students, graduate students, or students enrolled in professional programs.

(e) The board shall prescribe requirements for an institution of higher education to provide on the institution's Internet website consumer-friendly and readily

understandable information regarding student financial aid opportunities. The required information must be provided in connection with the information displayed under Subsection (c)(1) and must include a link to the primary federal student financial aid Internet website intended to assist persons applying for student financial aid.

(f) The board shall provide on the board's Internet website a program or similar tool that will compute for a person accessing the website the estimated net cost of attendance for a full-time entering first-year student attending an institution of higher education. The board shall require each institution to provide the board with the information the board requires to administer this subsection.

(g) The board shall prescribe the initial standards and requirements under this section not later than January 1, 2010. Institutions of higher education shall comply with the standards and requirements not later than April l, 2010. This subsection expires January 1, 2011.

(h) The board shall encourage private or independent institutions of higher education approved under Subchapter F to participate in the tuition equalization grant program, to the greatest extent practicable, to prominently display the information described by Subsections (a) and (b) on their Internet websites in accordance with the standards established under those subsections, and to conform to those standards in electronic and printed materials intended to provide to prospective undergraduate students information regarding the cost of attendance at the institutions. The board shall also encourage those institutions to include on their Internet websites a link to the primary federal student financial aid Internet website intended to assist persons applying for student financial aid.

(i) The board shall make the program or tool described by Subsection (f) available to private or independent institutions of higher education described by Subsection (h), and those institutions shall make that program or tool, or another program or tool that complies with the requirements for the net price calculator required under Section 132(h)(3), Higher Education Act of 1965 (20 U.S.C. Section 1015a), available on their Internet websites not later than the date by which the institutions are required by Section 132(h)(3) to make the net price

calculator publicly available on their Internet websites.

SECTION 4. Section 51.974, Education Code, as added by this Act, applies beginning with the 2010 fall semester.

SECTION 5. As soon as practicable after the effective date of this Act, each public institution of higher education shall establish an online list of work-study employment opportunities for students as required by Section 56.080, Education Code, as added by this Act.

SECTION 6. This Act takes effect immediately if it receives a vote of two-thirds of all the members elected to each house, as provided by Section 39, Article III, Texas Constitution. If this Act does not receive the vote necessary for immediate effect, this Act takes effect September 1, 2009.

_____ _____
President of the Senate Speaker of the House

I certify that H.B. No. 2504 was passed by the House on May 8, 2009, by the following vote: Yeas 138, Nays 0, 2 present, not voting; and that the House concurred in Senate amendments to H.B. No. 2504 on May 29, 2009, by the following vote: Yeas 143, Nays 0, 1 present, not voting.

_____
Chief Clerk of the House

I certify that H.B. No. 2504 was passed by the Senate, with amendments, on May 27, 2009, by the following vote: Yeas 31, Nays 0.

_____
Secretary of the Senate
APPROVED: _____
Date

_____
Governor

1.11 Scholastic Dishonesty

Any scholastic dishonesty will be referred to the Dean of Students Office. The following passage is taken from the University of Texas at Austin Information Handbook for Faculty.

The Discipline Policies Committee believes that in most cases of scholastic dishonesty the student forfeits the right to credit in that course, and that a penalty of "F" for the course may be warranted. In addition to the academic penalties assigned by a faculty member, the Dean of Students or the hearing officer may assign one or more of the University discipline penalties listed in the "General Information" bulletin, Appendix C, Sections 11-501 and 11-502. Certain types of misconduct, such as a student substituting for someone else on an exam or having someone substitute for the student, submitting a purchased term paper, or altering academic records, have usually involved a penalty of suspension from the University.

As a reminder, the "UT Code of Conduct" is available (`http://catalog.utexas.edu/general-information/the-university/#universitycodeofconduct`) where plagiarism, cheating, and other issues are described. If there are any questions, please see the UT General Information document about the Academic Policies and Procedures of UT Austin (`http://catalog.utexas.edu/general-information/academic-policies-and-procedures/`).

We fully support the University's scholastic honesty policies, and we will follow the University's policies in the event of any scholastic dishonesty. If you are ever unsure whether some act would be considered in violation of the University's policies, do not hesitate to ask your instructors or other University academic representatives.

1.12 Students with Disabilities

Students with disabilities (`http://ddce.utexas.edu/disability/`) may request appropriate academic accommodations from the Division of Diversity and Community Engagement, Services for Students with Disabilities, 512-471-6259.

1.13 Religious Holidays

A notice regarding accommodations for religious holidays. By UT Austin policy, you must notify your instructor(s) of your pending absence at least fourteen days prior to the date of observance of a religious holy day. If you must miss a class, an examination, a work assignment, or a project in order to observe a religious holy day, you will be given an opportunity to complete the missed work within a reasonable time after the absence.

1.14 Emergency Evacuation

The following recommendations regarding emergency evacuation from the Office of Campus Safety and Security, 512-471-5767, or see the safety office website (`http://www.utexas.edu/safety/`).

Although not likely pertinent for on-line courses, occupants of buildings on The University of Texas at Austin campus are required to evacuate buildings when a fire alarm is activated. Alarm activation or announcement requires exiting and assembling outside. Familiarize yourself with all exit doors of each classroom and building you may occupy. Remember that the nearest exit door may not be the one you used when entering the building. Students requiring assistance in evacuation shall inform their instructor in writing during the first week of class. In the event of an evacuation, please follow the instruction of faculty or class

instructors. Do not re-enter a building unless given instructions by one of the following: Austin Fire Department, The University of Texas at Austin Police Department, or the UT Fire Prevention Services office.

Information regarding emergency evacuation routes and emergency procedures is available (`http://www.utexas.edu/emergency`).

1.15 UT Required Notices

The University of Texas (UT) requires that we provide a significant amount of information about the organization, operation, and grading of our course. We believe this document provides the information required by UT; please let us know if we need to include something more.

2 Basic Logic Review

This is going to be a quick review of some content you should have encountered in your previous courses. In particular, the following words should invoke some (hopefully pleasant) memories:

- Axiomatic Logic Systems
- Propositional Logic
- Properties of a Logic
- Natural Deduction
- Predicate Logic
- Proof Techniques

2.1 Axiomatic Logic Systems

All axiomatic logic systems have three components – inference rules, axioms, and theorems. Both inference rules and axioms are assumed. Theorems are proved from axioms using inference rules. From a computational systems perspective, the inference rules process axioms as input and produce theorems as output. There is a strong analogy one can draw between traditional computational systems and axiomatic logic systems. In the same way that a processor executes program statements with inputs to produce outputs, a prover (human or machine) uses inference rules with axioms to produce theorems.

In a conventional computational system, placement of the hardware/software boundary is a design decision. Any given computational task can be implemented either in hardware or in software. The tradeoff in such systems is usually between speed of execution and flexibility. Usually, a task implemented in hardware executes faster than if it is implemented in software. However, once implemented in hardware a task is more difficult to modify or extend than if it is implemented in software. One goal of RISC design is to simplify the hardware by moving tasks from hardware to software. For example, CISC machines provide complex addressing modes with hardware circuits to compute array cell addresses. The equivalent address computation is done in software in a RISC machine.

A similar design decision exists in axiomatic logic systems with the placement of the inference rule/axiom boundary. It is possible to have two different logic systems produce equivalent sets of theorems but with different sets of inference rules and axioms. What is an inference rule in one system might be a corresponding theorem or axiom in the other. The tradeoff is more subjective in logic systems, as there is apparently no metric of goodness that can be quantified as objectively as can the speed of execution in computational systems. It can be harder to prove that an inference rule is sound than it is to prove that an axiom is sound. Deductive systems often arrange for fewer inference rules to make the soundness proof easier.

This lecture presents a logic system that places the boundary between inference rules and axioms to minimize the number of inference rules. We maintain that the primary advantage of such a system is a human one. That is, manual proofs in such systems are easier to understand and to design than in other systems.

This lecture borrows heavily on material from the textbook by Gries and Schneider *A Logical Approach to Discrete Math* (`https://www.cs.cornell.edu/info/people/gries/Logic/`

`LogicalApproach.html`). The paper by Warford, Vega and Staley *A Calculational Deductive System for Linear Temporal Logic* (`https://dl.acm.org/doi/10.1145/3387109`) builds directly on the work of Gries and Schneider and is also the source of much of this lecture material.

2.2 Propositional Logic

Propositional calculus is a formal system of logic based on the unary operator negation $\neg$, the binary operators conjunction $\wedge$, disjunction $\vee$, implies $\Rightarrow$ (also written $\rightarrow$), and equivalence $\equiv$ (also written $\leftrightarrow$), variables (lowercase letters $p, q, \ldots$), and the constants *true* and *false*. Hilbert-style logic systems, H, are the deductive logic systems traditionally used in mathematics to describe the propositional calculus. Typical of such descriptions with applications to computer science is the text by Ben-Ari cite(Ben). A key feature of such systems is their multiplicity of inference rules and the importance of modus ponens as one of them.

In the late 1980's, Dijkstra and Scholten cite, and Feijen cite developed a method of proving program correctness with a new logic based on an equational style. This equational deductive system, E, is the basis of books by Kaldewaij cite(Kald) and Cohen cite(Cohen). In contrast to H systems, E has only four inference rules – Substitution, Leibniz, Equanimity, and Transitivity. In E, modus ponens plays a secondary role. It is not an inference rule, nor is it assumed as an axiom, but instead is proved as a theorem from the axioms using the inference rules.

Gries and Schneider cite(Gries1995, Gries1995145) show that E, also known as a *calculational* system, has several advantages over traditional logic systems. The primary advantage of E over H systems is that the calculational system has only four proof rules, with inference rule Leibniz as the primary one. Roughly speaking, Leibniz is "substituting equals for equals," hence the moniker *equational* deductive system. In contrast, H systems rely on a more extensive set of inference rules. We find proofs in E easy to understand and to teach, because the substitution of equals for equals is common in elementary algebraic manipulations.

Another major advantage of E over H systems is the sequential format of its proof syntax. Proofs in H systems have a bottom-up tree structure, which is sequentialized with multiple references to previously numbered lines. For example, a proof of formula f_2 might begin by establishing the validity of a formula f_1 on lines 1 through 4. Then, on lines 5 through 9, it might establish the validity of $f_1 \Rightarrow f_2$. Then, on line 10, it would refer back to lines 4 and 9 and invoke modus ponens to establish the validity of f_2.

In contrast, proofs in E have a top-down structure and proceed sequentially with each step self-contained. There is no need to number the lines in a proof in E because reference is never made to a previous intermediate step of the proof. Instead, each line depends only on the immediately preceding line by invoking a previously-proved theorem or an axiom.

There is an analogy between the proof style of H systems versus the proof style of E, and the unstructured " *goto* " style of programming versus structured programming. In the same way that the goto statement can produce spaghetti code that is more difficult to understand than structured code, proofs in H systems are more difficult to understand than proofs in E. It is perhaps not coincidental that Dijkstra, who ignited the goto controversy with his famous CACM letter cite(Dijkstra:1968:LEG:362929.362947), was the prime developer of E.

Proof syntax is no guarantee of clarity. In the same way that a well-written assembly language program can be easier to understand than a poorly-written program in a structured high-order language, a well-written proof in H can be easier to understand than a poorly-written proof in E.

We agree with Gries and Schneider cite(LADM) that, "We need a style of logic that can be used as a tool in every-day work. In our experience, an equational logic, which is based on equality and Leibniz's rule for substitution of equals for equals, is best suited for this purpose." These advantages of E over H systems are primarily *human* advantages, not necessarily machine advantages. That is, the motivation behind this work is based on teaching and human understanding, as opposed to machine theorem provers or proof assistants.

In 1994, Gries and Schneider published *A Logical Approach to Discrete Math* (LADM) cite(LADM), in which they first develop E for propositional and predicate calculus, and then extend it to a theory of sets, a theory of sequences, relations and functions, a theory of integers, recurrence relations, modern algebra, and a theory of graphs. Using calculational logic as a tool, LADM brings all the advantages of E to these additional knowledge domains.

Another excellent source of information on these topics can be found at *An Introduction to teaching logic as a tool* (`https://www.cs.cornell.edu/home/gries/Logic/Introduction.html`). This a web-site set up and managed by Gries and Schneider.

Here are some review questions.

1. Recall that a formal logical system has four parts
   1. a set of symbols,
   2. a set of formulas constructed from the symbols,
   3. a set of distinguished formulas, call axioms, and
   4. a set of inference rules.

   What distinguishes theorems from axioms? How do you prove that a formula of the logic is a theorem?
2. For the equational logic E
   1. the set of symbols are $(,), =, \neq, \equiv, \not\equiv, \neg, \vee, \wedge, \Rightarrow, \Leftarrow$, the constants *true* and *false*, and boolean variables $p, q, \ldots$
   2. the set of formulas are constructed from these symbols, (e.g., $p \vee q, p \wedge q, \neg p \vee p$)
   3. the set of distinguished formulas, called axioms, contains 15 elements which are identified on the available equation sheet, and
   4. the set of four inference rules: (I1) Substitution, (I2) Leibniz, (I3) Equanimity, and (I4) Transitivity.

   The theorems of E are the formulas that are shown to be equivalent to an axiom using the inference rules. Some of the theorems of E are listed in the equation sheet handout. How many theorems are there in E?
3. One can also define an axiomatic system for propositional calculus with the following (minimal?) foundation.
   1. the set of symbols are $(,)\neg, \Rightarrow$, the constants *true* and *false*, and boolean variables $p, q, \ldots$

2. the set of formulas are constructed from these symbols, (e.g., $p \Rightarrow q, p \Rightarrow \neg q, \neg p$)
3. the set of distinguished formulas, called axioms, contains 3 elements which are as follows, and
   - Ax1. $p \Rightarrow (q \Rightarrow p) \ldots (4.1)$
   - Ax2. $(p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)) \ldots (3.64)$
   - Ax3. $(\neg p \Rightarrow \neg q) \Rightarrow (q \Rightarrow p) \ldots (3.61)$
4. the set of one inference rule (Modus Ponens): $\frac{\mathbf{P},\ \mathbf{P \Rightarrow Q}}{\mathbf{Q}} \ldots (3.77)$.

Can you find another propositional calculus system that is smaller (fewer axioms, fewer inference rules) than this system? Can you find who is credited with first presenting this system? Create definitions for the logical connectives $\vee, \wedge, \equiv$.

- Def. $p \equiv q$?
- Def. $p \vee q$?
- Def. $p \wedge q$?

2.3 Properties of a Logic

The following are some often discussed properties of a logic. We will not go into these topics in cs340d, but list them here for your reference and follow-up investigation.

- **Consistent**: a logic is consistent if at least one formula is a theorem, and at least one formula is not a theorem.
- **Interpetation**: an interpretation of a logic is the assignment of meaning to operators, constants and variables of a logic.
- **Model**: an interpretation is a model if and only if every theorem is mapped to *true* by the interpretation.
- **Sound**: a logic is sound if every theorem is valid.
- **Complete**: a logic is complete if every valid formula is a theorem.
- **Decision Procedure**: a decision procedure for a logic is an algorithm that determines the validity of a formula in the logic. Given, as we will see in the material on truth-tables coming up, that the decision procedure could require checking 2^n different cases decision procedures typically will return *true* or *false* or that it does not have the resources to determine the answer. A data structure, called a binary decision diagram, is often used to represent a boolean function for the purposes of computing its validity or satisfiablity.

2.4 Natural Deduction

Natural deduction is a Hilbert-style propositional logic due to Gerhard Gentzen. Natural deduction has no axioms, but instead, has two inference rules for each operator and constant (e.g., $\equiv, \neg, \vee, \wedge, true \ldots$). One rule introduces the symbol into a theorem and one rule eliminates the symbol from a theorem.

Since we have just spent some time above arguing for the superiority of the logic E, we will not go further into natural deduction, except to invite the student to look into proofs in H and decide for themselves on an approach to proofs. Set your search engine looking for David Hilbert, Gerhard Gentzen, ND, deduction, natural deduction, Hilbert-style, proof theory, modus ponens, inference rules and so on.

2.5 Predicate Logic

As we have seen, propositional logic reasons with *boolean* variables and *boolean* operators. Sometimes it's useful to talk about propositions whose truth value depends on boolean functions whose *arguments* are of types other than boolean. For example consider a function called evenp(i) where i is an integer and evenp(i) returns T if i is even and F otherwise.

Objects, such as evenp, are called predicates. Predicates are functions which map custom domains onto a boolean range. Predicate logic extends propositional logic to use these *functions*.

To deal with the extent of the newly introduced predicates, (e.g., the set of i for which evenp(i) = T is infinite), predicate logic has the additional concepts of *universal* quantification and *existential* quantification which increase reasoning power and expressibility. These are written as follows.

$(\forall x \mid R : P)$ and is read "for all x such that R holds, P holds".

$(\exists x \mid R : P)$ and is read "there exists an x in the range R such that P holds".

This is as far as we will take this topics in cs340d for now. The students are ecouraged to look more into the literature of this topic as need and interest dictates.

Here are some review questions.

1. Predicate logic allows us to make statements about sets of objects. Write the predicate logic formulas for the following claims.
   - All prime numbers greater than 2 are odd numbers.
   - All cs340d students are smart, happy and love zoom meetings.
   - If it is Tuesday or Thursday, then at 9:30AM cs340d students are in a zoom meeting.
   - There is no Real Number x for which $x^2 + 1 = 0$.
2. (from LADM) Let the two-place predicate $L(x, y)$ mean x loves y. Write the following English sentences in predicate logic.
   - Everybody loves somebody.
   - Somebody loves somebody.
   - Everybody loves everybody.
   - Nobody loves everybody.
   - Somebody loves nobody.

2.6 Proof Techniques

Now we do some proofs in the equational logic E. Recall a formal logical system has the following parts, with the parts for E shown as a particular example.

1. **a set of symbols**, which for E are: $(,), =, \neq, \equiv, \not\equiv, \neg, \vee, \wedge, \Rightarrow, \Leftarrow$, the constants *true* and *false*, and boolean variables $p, q, \ldots$
2. **a set of formulas constructed from these symbols**, which for E include formula such as (e.g., $p \vee q \Rightarrow, p \wedge q \Rightarrow p, \neg p \vee p$)
3. **a set of distinguished formulas**, called axioms, which for E contains 15 elements identified on the available equation sheet, and
4. **a set of inference rules**, which for E are: (I1) Substitution, (I2) Leibniz, (I3) Equanimity, and (I4) Transitivity.

2.6.1 Proving Axioms

Axioms are formulas in the logic that are accepted as valid without proof. However, they still have to be true. If you can find one counterexample (assignment of a truth value to each variable for which the axiom becomes false) for the axiom, it has to be dropped. The validity of axioms can be established by appeal to intuition, appeal to a semantic model of the system or elaboration of a truth-table. A truth-table shows the value of a boolean expression for all values of it's input varibles. If the formula is *true* under all conditions it is said to be valid (also called a tautology).

For a propositional formula of 2 variables p, q all possible combinations of p and q would create a truth-table structure as follows. In general a truth-table of n boolean variables will have 2^n rows.

p	**q**	**propositional formula**
T	T	?
F	T	?
T	F	?
F	F	?

Use the next two templates and construct truth-tables for confirming that Axioms (3.2) and (3.3) are valid.

(3.2) Axiom, Symmetry of $\equiv$

p	**q**	$p \equiv q$	$q \equiv p$	$p \equiv q \equiv q \equiv p$
T	T	T	T	?
F	T	F	F	?
T	F	F	F	?
F	F	T	T	?

(3.3) Axiom, Identity of $\equiv$

p	**q**	*true*	$q \equiv q$	$true \equiv q \equiv q$
T	T	T	T	?
F	T	T	T	?
T	F	T	T	?
F	F	T	T	?

Now with the examples of confirming axioms 3.2 and 3.3 in hand, construct the truth-table to confirm the validity of axiom 3.1.

(3.1) Axiom, Associativity of $\equiv$: $((p \equiv q) \equiv r) \equiv (p \equiv (q \equiv r))$

p	**q**	**r**	$(p \equiv q)$	$(p \equiv q) \equiv r$	**(axiom) proposition**	$p \equiv (q \equiv r)$	$(q \equiv r)$
T	T	T					
F	T	T					
T	F	T					
F	F	T					
T	T	F					
F	T	F					
T	F	F					
F	F	F					

Finally, for the student who truly loves truth-table construction, prove the following formula in E is a theorem using truth-tables. A template for 4 boolean variables is shown below.

(3.77.3) $((p \Rightarrow (q \Rightarrow r)) \wedge (r \Rightarrow s)) \Rightarrow (p \Rightarrow (q \Rightarrow s))$

p	**q**	**r**	**s**	**propositional formula (3.77.3)**
T	T	T	T	
F	T	T	T	
T	F	T	T	
F	F	T	T	
T	T	F	T	
F	T	F	T	
T	F	F	T	
F	F	F	T	
T	T	T	F	
F	T	T	F	
T	F	T	F	
F	F	T	F	
T	T	F	F	
F	T	F	F	
T	F	F	F	
F	F	F	F	

Here is an exercise for the interested student.

The last truth-table with 4 variables, and 16 rows is quite tedious most would agree. But such a problem is tiny from an industrial perspective. You could still do the following formula with 5 variables and 32 rows with truth-tables, but instead take a look at using ACL2 to convince yourself that (3.77.2) is a theorem.

(3.77.2) $((p \Rightarrow q) \Rightarrow (r \Rightarrow s)) \wedge (s \Rightarrow t) \Rightarrow ((p \Rightarrow q) \Rightarrow (r \Rightarrow t))$

p	**q**	**r**	**s**	**t**	**propositional formula (3.77.2)**
T	T	T	T	T	
F	T	T	T	T	
T	F	T	T	T	
F	F	T	T	T	
T	T	F	T	T	
F	T	F	T	T	
T	F	F	T	T	
F	F	F	T	T	
T	T	T	F	T	
F	T	T	F	T	
T	F	T	F	T	
F	F	T	F	T	
T	T	F	F	T	
F	T	F	F	T	
T	F	F	F	T	
F	F	F	F	T	
T	T	T	T	F	
F	T	T	T	F	
T	F	T	T	F	
F	F	T	T	F	
T	T	F	T	F	
F	T	F	T	F	
T	F	F	T	F	
F	F	F	T	F	
T	T	T	F	F	
F	T	T	F	F	
T	F	T	F	F	
F	F	T	F	F	
T	T	F	F	F	
F	T	F	F	F	
T	F	F	F	F	
F	F	F	F	F	

2.6.2 Inference Rules of *E*

Here we list the inferrence rules of E. For more information see section 2.1.2 of *A Calculational Deductive System for Linear Temporal Logic* (`https://dl.acm.org/doi/10.1145/3387109`).

(I1) **Substitution** : $\frac{\mathbf{E}}{\mathbf{E[z:=F]}}$

(I2) **Leibniz** : $\frac{\mathbf{X=Y}}{\mathbf{E[z:=X]=E[z:=Y]}}$

(I3) **Equanimity** : $\frac{\mathbf{X},\ \mathbf{X{=}Y}}{\mathbf{Y}}$

(I4) **Transitivity** : $\frac{\mathbf{X{=}Y},\ \mathbf{Y{=}Z}}{\mathbf{X{=}Z}}$

2.6.3 Direct Proof

Direct proofs are often concerned with proving conditionals; statements of the form $P \Rightarrow Q$. Since the truth-table of a conditional tells us that if P is *false*, then $P \Rightarrow Q$ is *true*, direct proof is focused on showing that Q **must be** *true* if P is *true*.

This form of proof is also called deduction. We state the proof strategy as follows.

To prove $P_1 \wedge P_2 \wedge \ldots \Rightarrow Q$ assume $P_1, P_2, \ldots$ and prove Q.

Ok, let's do some direct proofs.

- Prove (3.4) *true* is a theorem.
- Prove (3.5) $p \equiv p$ (Reflexivity of $\equiv$)
- Prove (3.59) $p \Rightarrow q \equiv \neg p \vee q$ (Implication)
- Prove (3.77) $p \wedge (p \Rightarrow q) \Rightarrow q$ (Modus Ponens)

2.6.4 Mutual Implication Proof

To prove $P \equiv Q$, prove $P \Rightarrow Q$ and $Q \Rightarrow P$. This proof strategy is justified by metatheorem (4.7) and theorem (3.80) Mutual implication.

Ok, let's do some Mutual implication proofs.

- Prove (3.15) $\neg p \equiv p \equiv false$ (This is better handled as a direct equivalence proof)
- Prove (3.81) $(p \Rightarrow q) \wedge (q \Rightarrow p) \Rightarrow (p \equiv q)$ (Antisymmetry)
- Prove (141) $p\, U [] p \equiv [] p$ (Absorption o f U into $[]$)

2.6.5 Truth Implication Proof

To prove P, prove $true \Rightarrow P$. This proof strategy is justified by metatheorem (4.7.1) and theorem (3.73) Left identity of $\Rightarrow$.

Ok, let's do some Truth implication proofs.

- Prove (27) $p \wedge \neg p\, U q \Rightarrow q$
- Prove (142) $p\, U (q \wedge r) \Rightarrow p\, U (q\, U r)$ (Right $\wedge U$ strengthening)
- Prove (193) $(p \Rightarrow q)\, W p$

2.6.6 Proof by Contradiction

To prove P, prove $\neg P \Rightarrow false$. This proof strategy is justified by metatheorem (4.9) and theorem (3.74.1) $\neg P \Rightarrow false \equiv P$

Ok, let's do some Proof by Contradiction proofs.

- Prove (92) $\Diamond p \wedge [] \neg p \equiv false$ ($\Diamond$ contradiction)
- Prove (165) $[]((p \vee [] q) \wedge ([] p \vee q)) \equiv [] p \vee [] q$

2.6.7 Proof by Contrapositive

To prove $P \Rightarrow Q$, prove $\neg Q \Rightarrow \neg P$. This proof strategy is justified by metatheorem (4.12) and theorem (3.61) Contrapositive.

Ok, let's do some Proof by Contrapositive proofs.

- Prove (57) $[](p \Rightarrow \circ p) \Rightarrow (p \Rightarrow [] p)$ ([] induction)
- Prove (58) $[](\circ p \Rightarrow p) \Rightarrow (\diamond p \Rightarrow p)$ ($\diamond$ induction)
- Prove (75) $p \wedge \diamond \neg p \Rightarrow \diamond(p \wedge \circ \neg p)$
- Prove (75) is equivalent to (57) [] induction

2.6.8 Proof by Case Analysis

A proof by case analysis is based on the following theorem.

(4.6) $(p \vee q \vee r) \wedge (p \Rightarrow s) \wedge (q \Rightarrow s) \wedge (r \Rightarrow s) \Rightarrow s$

In general, a case analysis proof is not recommended. Therefore we will not cover it further here. But the student should know that such a technique exists and they can explore it on their own as needed.

2.6.9 Mathematical Induction

Mathematical induction is particularly useful when you want to prove countably many statements that share a similar "form". For example, legend has it that Carl Friedrich Gauss proved the following identity as a very young boy:

$$1 + 2 + \cdots + 100 = 100(100 + 1)/2.$$

Generalising, this is

$$\forall n \in N,\ 1 + 2 + \cdots + n = n(n + 1)/2,$$

which is really the following countably infinite statements

$$[1 = 1(1 + 1)/2] \wedge [1 + 2 = 2(2 + 1)/2] \wedge \cdots \wedge [1 + 2 + \cdots n = n(n + 1)/2] \wedge \cdots$$

With only the tools we've discussed up to now, proving each of these statements would involve verifying each of these expressions by hand, which would take a very long time and would be very annoying. Mathematical induction gives us a "shortcut".

There are two (equivalent) forms of mathematical induction. Let's talk about *weak induction* first. Let P_k denote a statement with k varying over the naturals. Weak induction says that if we can just prove two particular statements, then P_k would be true for all naturals k. The two statements are:

- P_1 is true (base case);
- $P_k \rightarrow P_{k+1}$ is true (inductive step).

So now something that has been would have taken literally forever to prove has been boiled down into proving just two simple statements. This is so powerful, it's almost like cheating. Of course, to say "two simple statements" might be a bit disingenuous. While P_1 is usually simple enough, showing $P_k \rightarrow P_{k+1}$ is usually a bit trickier. Luckily, *strong induction* can make proving the inductive step a lot easier, making induction even more unfairly overpowered. Strong induction says that if you can prove the following two statements, then you have proven P_k for all naturals k:

- P_1 is true (base case);

- $(P_1 \wedge P_2 \wedge \cdots \wedge P_k) \rightarrow P_{k+1}$ is true (inductive step).

The only difference between strong induction and weak induction is that you have a lot more "ammo" for proving the inductive step.

If you haven't seen these definitions of induction before, don't worry. As long apply induction correctly to other problems, everything will be fine. As a test, make sure you can follow our proof for the sum of n natural numbers. We will only use weak induction. Recall that our statement, P_k, is now

$$\sum_{n=1}^{k} n = k(k+1)/2,$$

and, according to weak induction, it is sufficient to prove P_1 and $P_k \rightarrow P_{k+1}$, which is exactly what we'll do.

- $P_1 \equiv 1 = 1(1+1)/2$:
  Observe that $1 = 2/2 = 1(2)/2 = 1(1+1)/2$.
- $P_k \rightarrow P_{k+1} \equiv (1+2+\cdots+k = k(k+1)/2) \rightarrow ((1+2+\cdots+k+k+1) = (k+1)(k+2)/2)$:
  Notice that the first k terms of the LHS of P_{k+1} is equivalent to the LHS of P_k, which validates the following substitution:

  $$1 + 2 + \cdots + k + (k+1) = k(k+1)/2 + (k+1)$$

  Then, using what we know about fractions and quadratics, we get

  $$k(k+1)/2 + (k+1) = [k(k+1) + 2(k+1)]/2 = [k^2 + 3k + 2]/2 = (k+1)(k+2)/2,$$

  which completes the inductive step.

Here are some exercises for your enjoyment:

1. Reform our proof of the sum of n natural numbers to use strong induction instead of weak induction.
2. Show that the two definitions of induction are equivalent.
3. Prove that the following program sets i to n:

```
i = 0
while i < n :
  i = i + 1
```

2.7 Review of Linear Temporal Logic

In this section we provide a brief overview of the basics of Linear Temporal Logic (LTL). It is recommended that the student read the paper by Warford, Vega and Staley *A Calculational Deductive System for Linear Temporal Logic* (`https://dl.acm.org/doi/10.1145/3387109`) prior to the class lecture on this topic. This paper is freely available for download on the ACM website. The paper is tutorial in nature and does not assume any prior experience with LTL. It does however, assume some proficiency in proving theorems in propositional calculus using the system E which was introduced in an earlier section titled Review of Basic Logic. Also the document *vega-equations-new.pdf* will be made available to anyone interested. This document is a collection of a large number of LTL theorems that were collected in work on a survey of the LTL literature. Here are the topics to be covered on LTL.

- Axiomatic Logic System for LTL

- Stating Properties in LTL
- Temporal Deduction
- Proof techniques and Proofs
- How to Prove it - Tips
- Example: Program Properties and a Proof

2.7.1 Axiomatic Logic System for LTL

We will do our LTL proofs in the equational logic E for propositional calculus, extended for LTL. Recall a formal logical system has the following parts, with the parts for E shown as a particular example.

1. **a set of symbols**, which for E are: $(,), =, \neq, \equiv, \not\equiv, \neg, \vee, \wedge, \Rightarrow, \Leftarrow$, the constants *true* and *false*, and boolean variables $p, q, \ldots$
2. **a set of formulas constructed from these symbols**, which for E includes formula such as (e.g., $p \Rightarrow p \vee q,\ p \wedge q \Rightarrow p,\ \neg p \vee p$)
3. **a set of distinguished formulas**, called axioms, which for E contains 15 elements identified on the available equation sheet, and
4. **a set of inference rules**, which for E are: (I1) Substitution, (I2) Leibniz, (I3) Equanimity, and (I4) Transitivity.

To this logic machinery we add the following to include LTL in E.

1. **the additional symbols**: $\circ, \diamond, [], U, W$ These symbols are the operators of LTL. There are 3 unary operators: the *next* operator $\circ$, the *eventually* operator $\diamond$ and the *always* operator $[]$. There are two binary operators: the *until* operator U and the the *wait* operator W.
2. **the additional formulas** that can be constructed with the new temporal operators denoted by the symbols added above. (e.g. $\circ p \equiv \neg \circ \neg p,\ \diamond p \equiv p \vee \circ \diamond p,\ p\, U q \Rightarrow \diamond q,\ [] p \Rightarrow p\, W q$)
3. **the additional axioms and definitions** used to define the behavior of the temporal logic operators are added to the axiom set of E. LTL adds 14 axioms of behavior, and 3 definitions of operators to the existing **set of distinguished formulas**. This brings the total for the combined set of propositional logic and LTL to 32 formulas for E.
4. there are no additions to the **set of inference rules** (which was a key goal of the work).

2.7.2 Stating Properties in LTL

As we have discussed throughout this course being able to specify intended program behavior in a precise way (read mathematical and formal way) is a key enabler to program design. But how are we to be sure our programs, (which may, e.g. be embedded in a pacemaker, a nuclear power plant, a financial application, or an autonomous vehicle) will behave as intended?

First, we must say what we intend in a specification language expressive enough to define program behavior. And second, we must be able to prove that what we have created (the program) satisfies all the required behaviors. This is accomplished through the selection of a logic and a proof system respectively.

For the domain of concurrent programming, Amir Pneuli is generally credited with introducing the use of LTL for formal verification in 1977. Using LTL, a specification is a set of

properties, expressed as LTL formulas, which must be satisfied by every possible behavior of the implementation. This formal specification then, in the next step of the engineering process, supports a robust debugging and verification process leading to creation of a high quality product.

Most documents defining the requirements for a software-intensive system, if they exist at all, are written in natural language. While natural language is expressive and nuanced, it is also imprecise, ambiguous and often verbose. On the other hand, formal languages are precise, but not very expressive.

It appears that LTL has passed the test of time. Since its introduction for use in program verification in 1977, it has become a widely used tool in academia and industry. As an example LTL is used in the following systems: SPIN, MAUDE, SPOT, PVS, Isabelle, Formal Check and this list is by no means exhaustive. The approach to program verification using LTL is conceptually straigtht-forward. Write program requirements as a conjunction of LTL formulas that comprise the specification. Show that each formula is valid over the program. This can be done for each LTL formula expressing a property, one-by-one. Next we look at the kinds of properties that are often specified for concurrent programs.

There are two often used categories of LTL property formulas: **safety** properties and **liveness** properties.

Safety properties are properties of the form $[]p$. They are often used to express an invariance of some state property over all computations. They are commonly used to say "something bad" does not happen. For example they could express non-termination of a concurrent program using a formula such as $[]\neg HALT$ in their specification.

A safety property can also be a precedence constraint. For example, one might want to require that if some event q happens it is preceded by event p. Let q be the predicate $(y = 2)$ and p be the predicate $(x = 1)$, then the LTL formula $(y \neq 2)W(x = 1)$ specifies that the negation of q either always holds or holds until p does, after which time q holds.

Examples of typical safety properties include

- Global invariants: $[](p \Rightarrow []p)$, which can be read as "once p, always p."
- Partial correctness: $p \Rightarrow [](HALT \Rightarrow q)$, where p is the pre-condition to running the program, and q is the post-condition.
- Deadlock freedom: $[]\neg HALT$
- Mutual exclusion: $[]\neg(CS_1 \wedge CS_2)$, where CS_n means process n is in the *critical section*.
- Wellformedness of data structures

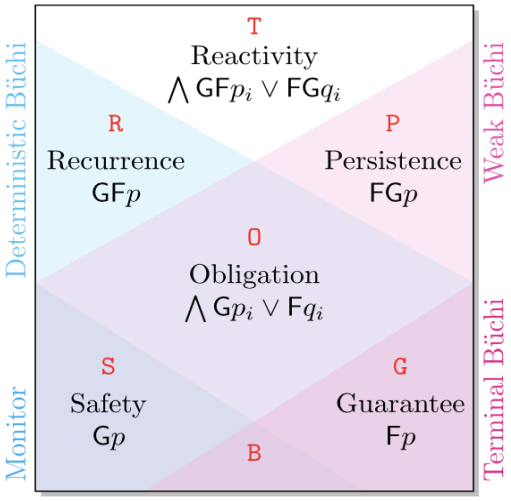
A liveness property states that "something good" eventually happens using a formula such as $\diamond q$.

Examples of typical liveness properties include

- Termination: $\diamond HALT$
- Starvation Freedom: $[](p \Rightarrow \diamond q)$
- Request-Grant: $[](p \Rightarrow \diamond q)$
- Request Until Grant: $[](p \Rightarrow p\, U q)$
- Fairness Requirements: (strong) $[] \diamond p \Rightarrow [] \diamond q$. Every process that is enabled infinitely often, get's its turn to run infinitely often when it is enabled.

There is another classification of LTL property specifications that is widely known, and very useful. It is called the temporal hierarchy of Manna and Pnueli and was first described in their 1990 paper, *A Hierarchy of Temporal Properties* (`ftp://www-cs.stanford.edu/cs/theory/amir/hierarchy.ps`). We finish this section by listing the classes of the Manna-Pnueli hierarchy and giving some representative examples of the types of LTL formula that express those properties. You will see many similarities and overlapw with the *safety-liveness* categories that preceded it.

- **Reactivity**: these properties are boolean combinations of recurrence and persistence properties. They are formulas of the form: $[] \diamond p \vee \diamond [] q$. This formula says that either there are infinitely many states where p holds, or there are finitely many states where q does not hold.
- **Recurrence**: these properties are the dual of Persistence. They are formulas of the form: $[] \diamond p$. They express the notion that the trace of p contains infinitely many p-positions. They are used in expressing properties of Justice and Fairness in LTL.
- **Persistence**: these properties are used to specify an eventual stabilization of a state or property of the system. Once the stabilization occurs it persists. Persistence properties are of the form $\diamond [] p$. Another example expressing persistence is $p \Rightarrow \diamond [] q$.
- **Obligation**: these properties are boolean combinations of safety and guarantee properties. They are formulas of the form: $[] p \vee \diamond q$ which is equivalent to $p \, W \diamond q$.
- **Safety**: these properties are often used to express an invariance of some state property over all computations. The negation of a safety property is a guarantee property. This can be shown, e.g. with the safety property $[] \neg BAD$. Its negation is $\neg [] \neg BAD$, which is equivalent to $\diamond BAD$ which is a guarantee property
- **Guarantee**: these properties are expressed by formulas of the form $\diamond p$. This formula states that at least 1 position in a computation satisfies p. Typically used to ensure that some event happens, e.g. termination. They are closest in meaning to the liveness class of formulas. An example guarantee property is $\diamond [(x = 1) \wedge \diamond (y = 2)]$



2.7.3 Temporal Deduction

The Deduction metatheorem (4.4) for propositional calculus from cite(LADM) can be extended in E for temporal deduction. This metatheorem (82) is stated in cite(Warford) as follows.

> (82) **Temporal Deduction**
>
> To prove $[]P_1 \wedge []P_2 \Rightarrow Q$, assume P_1 and P_2, and prove Q.
>
> You cannot use textual substitution in P_1 or P_2.

Temporal deduction is Theorem (2.1.6) of Kröger and Merz cite(Kröger), who also give the justification. Note that if you assume P in a step of an LTL proof of Q, you have not proved that $P \Rightarrow Q$, but rather that $[]P \Rightarrow Q$. We will see the application of this metatheorem in a later section where it is used in a LTL proof.

2.7.4 Proof techniques and Proofs in LTL

As we have seen, propositional logic reasons with *boolean* variables. Predicate logic includes reasoning with boolean functions over variables that are not necessarily themselves boolean variables. LTL extends reasoning power further by reasoning with variables that are traces (of finite or infinite length) of booleans that allow the truth-functional value of variables in LTL to change over time, depending on the evolving computations of a set of concurrent programs.

Similar to our approach to the axiomatic development of E for propositional calculus, we start our study of proving theorems in LTL with a look at proving axioms can be trusted to be *true*.

2.7.4.1 Proving Axioms in LTL

The issues are the same for accepting the axioms proposed for any LTL system as they were for the axioms of a propositional logic system. The axioms are formulas in the logic that are accepted as valid without proof. However they still have to be true. If you can find one counterexample (assignment of a truth value to each variable for which the axiom becomes false) for the axiom, it has to be dropped. The validity of axioms can be established by appeal to intuition, appeal to a semantic model of the system or elaboration of a truth-table.

In the case of LTL, the truth-table is a more complex object. A variable in LTL is not a boolean variable assigned one of two truth values, but a list or "trace" of boolean values assigned to represent the evolution of the variable's truth value over time. So for a boolean variable of propositional calculus, say p, its value is either *true* or *false*. For a variable of a LTL formula, it's value could take on an infinite number of values for infinite traces. In the case of finite traces a variable of trace length n could take on 2^n different traces. A truth-table shows the value of a boolean expression for all values of it's input variables. If the formula is *true* under all conditions it is said to be valid.

The following shows that the constant *true* evaluates to T in every state. And similarly for $false$.

LTL formula	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	...
true	T	T	T	T	T	T	T	T	T	...

$false$	F	F	F	F	F	F	F	F	F	...

The next truth-table shows that (54) Definition of [] (always) is a valid LTL formula. The last row of the truth-table shows the formula is always T. This LTL formula has only one temporal variable, p. For traces of length n, there would be 2^n different traces p could take on as a value. However, in most LTL systems traces are assumed to be infinitely long.

$[]p \equiv \neg \diamond \neg p$	s_0	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	...
p	T	F	F	T	F	T	T	T	T	...
$\neg p$	F	T	T	F	T	F	F	F	F	...
$[]p$	F	F	F	F	F	T	T	T	T	...
$\diamond \neg p$	T	T	T	T	T	F	F	F	F	...
$\neg \diamond \neg p$	F	F	F	F	F	T	T	T	T	...
$[]p \equiv \neg \diamond \neg p$	T	T	T	T	T	T	T	T	T	...

2.7.4.2 Direct Proof

Ok, let's do some direct proofs.

- Prove (83) Distributivity of $\wedge$ over U: $[]p \wedge q \; U r \Rightarrow (p \wedge q) \; U (p \wedge r)$
  This is a temporal deduction proof.
- Prove (153) Absorption of $[]\diamond$: $[]\diamond[]\diamond p \equiv []\diamond p$
- Prove (215) W induction $[](p \Rightarrow \circ p) \Rightarrow (p \Rightarrow p \; W q)$
- Prove (219) Absorption: $p \; W q \wedge q \equiv q$

2.7.4.3 Mutual Implication Proof

See this same section under Logic Review. Propositional calculus and LTL example proofs are listed together.

2.7.4.4 Truth Implication Proof

To prove P, prove $true \Rightarrow P$. This proof strategy is justified by metatheorem (4.7.1) and theorem (3.73) Left identity of $\Rightarrow$.

Ok, let's do some Truth implication proofs.

- Prove (254) Lemmon formula: $[]([]p \Rightarrow q) \vee []([]q \Rightarrow p)$

2.7.4.5 Proof by Contradiction

See this same section under Logic Review. Propositional calculus and LTL example proofs are listed together.

2.7.4.6 Proof by Contrapositive

See this same section under Logic Review. Propositional calculus and LTL example proofs are listed together.

2.7.4.7 Proof by Case Analysis

See this same section under Logic Review. Propositional calculus and LTL example proofs are listed together.

2.7.4.8 Mathematical Induction

Induction in E is handled implicitly by the structure of time in the logic. The Lemmon formula (254) imposes linearity of the time line, and the Dummett formula (S111) establishes the discreteness of time. Both of these formulas are theorems in the LTL we have presented. To give a feel for the difference in proving a theorem with induction as implicit, see the following proof of theorem (129).

(129) **Induction rule $\Box$:** $\Box(p \Rightarrow \bigcirc p \wedge q) \Rightarrow (p \Rightarrow \Box q)$

Proof:

$$
\begin{array}{ll}
 & \mathit{true} \\
= & \langle (55)\ \mathcal{U} \text{ induction with } r := \mathit{false} \rangle \\
 & \Box(p \Rightarrow (\bigcirc p \wedge q) \vee \mathit{false}) \Rightarrow (p \Rightarrow \Box q \vee q\ \mathcal{U}\ \mathit{false}) \\
= & \langle (11) \text{ Right zero of } \mathcal{U} \rangle \\
 & \Box(p \Rightarrow (\bigcirc p \wedge q) \vee \mathit{false}) \Rightarrow (p \Rightarrow \Box q \vee \mathit{false}) \\
= & \langle (3.30) \text{ Identity of } \vee,\ p \vee \mathit{false} \equiv p \text{ twice} \rangle \\
 & \Box(p \Rightarrow \bigcirc p \wedge q) \Rightarrow (p \Rightarrow \Box q) \quad \blacksquare
\end{array}
$$

An LTL proof with induction explicit, would follow a proof strategy like the one you see in the following proof of (S64). This structure is likely much more familiar to you. It follows

the format the we used in section (2.22.6.9) earlier in the proof of famous theorem of Gauss.

(S64) **Indefinite nested insertion:** $x_n \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-1} \,\mathcal{U}\, x_n \underbrace{)\ldots))}_{n-2 \text{ times}}$ for $n \geq 3$

Proof: By mathematical induction.

Base Case: $n = 3$. Prove $x_3 \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, x_3)$

Proof:

$$
\begin{aligned}
& \textit{true} \\
= \quad & \langle \text{(S62) Nested insertion with } p,q,r := x_1,x_2,x_3 \rangle \\
& x_3 \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, x_3) \quad \blacksquare
\end{aligned}
$$

Induction Step: Prove $x_n \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-1} \,\mathcal{U}\, x_n \underbrace{)\ldots))}_{n-2 \text{ times}}$

assuming $x_{n-1} \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-2} \,\mathcal{U}\, x_{n-1} \underbrace{)\ldots))}_{n-3 \text{ times}}$

as the induction hypothesis.

Proof: The proof is by (4.7.1) Truth implication.

$$
\begin{aligned}
& \textit{true} \\
= \quad & \langle \text{Assume the Induction Hypothesis is } \textit{true} \rangle \\
& x_{n-1} \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-2} \,\mathcal{U}\, x_{n-1} \underbrace{)\ldots))}_{n-3 \text{ times}} \\
= \quad & \langle \text{The above theorem with } x_{n-1} := x_{n-1} \,\mathcal{U}\, x_n \rangle \\
& x_{n-1} \,\mathcal{U}\, x_n \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-2} \,\mathcal{U}\, (x_{n-1} \,\mathcal{U}\, x_n) \underbrace{)\ldots))}_{n-3 \text{ times}} \\
\Rightarrow \quad & \langle \text{(29) } \mathcal{U} \text{ insertion with } q,p := x_n,x_{n-1} \\
& \quad \text{and (3.82a) Transitivity, } (p \Rightarrow q) \wedge (q \Rightarrow r) \Rightarrow (p \Rightarrow r) \rangle \\
& x_n \Rightarrow x_1 \,\mathcal{U}\, (x_2 \,\mathcal{U}\, (\ldots \,\mathcal{U}\, (x_{n-1} \,\mathcal{U}\, x_n \underbrace{)\ldots))}_{n-2 \text{ times}} \quad \blacksquare
\end{aligned}
$$

2.7.5 How to Prove it - Tips

Here is a collection of thought starters to keep you going as you try to prove a formula is a theorem. These are especially useful if the theorem is strongly resisting your best efforts. Use this list of questions and assertions as a checklist of ideas to ponder as you push to construct a successful proof.

- Are you sure the formula is a theorem? Why do you think so?

- Are you sure the formula is NOT a theorem? Can you prove it is NOT? Can you produce a counterexample? Remember it only takes 1 counterexample to kill a proposed theorem.
- Try all the different proof strategies you know: direct proof; mutual implication; truth implication; proof by contradiction; contrapositive; case analysis; mathematical induction; temporal deduction.
- Stay around the problem. Sleep on it. Visualize it. Play it like a movie in your mind's eye. Set it aside for a while, and come back later for a fresh attack.
- Can you prove formulas (syntactally) "close" to the one you want? *What can you prove*?
- Do parts of the formula look familiar? Can you devise a lemma approach to prove some supporting lemmas that will help you with proving the main formula?
- Get frustrated. It's ok, it means you are engaged, working on it and motivated.
- Can you use an existing automated theorm proving system, like ACL2, to prove the formula is a theorem?
- Can you do a simulation (or use model checking) to convince yourself that it is a theorem, and that you should keep going.
- Look for a missing axiom. Axiom sets can be wrong. Check to see if there is some logical structure that you know should exist, but does not follow from the axiom set. Add to, or modify the axiom set as required.
- Never give up. If you have tried all the above, get on the internet and see what else you can find out about the formula. If you are still convinced it is a theorem, go to the top and start again.

2.7.6 Example: Program Properties and a Proof

This example is taken from the internet. It is based on class notes by Dr. Alessandro Artale, Faculty of Computer Science, Free University of Bolzano, *Lecture III: Linear Temporal Logic* (`https://www.inf.unibz.it/~artale/FM/slide3.pdf`). While the program specification is from Dr. Artale, the formulation of the proof obligation and its proof are ours.

Problem Description:

A system has been created that should meet the following requirements, stated in LTL as follows.

$$[](Requested \Rightarrow \diamond Received)$$
$$[](Received \Rightarrow \circ Processed)$$
$$[](Processed \Rightarrow \diamond [] Done)$$

From the above show that it is **not** the case that the system continually re-sends a request, but never sees it completed ($[]\neg Done$). Another way to say this is that the statement

$$[]Requested \wedge []\neg Done$$

should be inconsistent.

Formulate a proof obligation for this system in E and prove the system meets the requirement.

First some questions for the student.

1. Place each of the three requirements above in the Manna-Pneuli hierarchy. You might find the following web page of some help in this task *SPOT: On-line Translator* (`https://spot.lrde.epita.fr/app/`).
2. Place the overall program correctness criteria, $[]p \Rightarrow \diamond s$, in the Manna-Pneuli hierarchy.

Solution:

We make the following abbreviations.

$$p \equiv Requested$$

$$q \equiv Received$$

$$r \equiv Processed$$

$$s \equiv Done$$

The system meets all of these requirements so we will say that the conjunction of the three requirements imply that the completion requirement ($Done$) is met. In our E with LTL we write the system requirements as

$$[](p \Rightarrow \diamond q) \wedge [](q \Rightarrow \circ r) \wedge [](r \Rightarrow \diamond [] s)$$

Now, if these are *true*, then the following

$$[]p \wedge []\neg s$$

should be *false*, or alternatively

$$\neg([]p \wedge []\neg s)$$

should be *true*. Since

$$\neg([]p \wedge []\neg s) \equiv ([]p \Rightarrow \diamond s),$$

we can state our proof obligation as follows:

$$[](p \Rightarrow \diamond q) \wedge [](q \Rightarrow \circ r) \wedge [](r \Rightarrow \diamond [] s) \Rightarrow ([]p \Rightarrow \diamond s)$$

Proof: (for the student to provide)

`(Example Proof)`

LTL Program property proof example for CS340d

(255) **example:** $\Box((p \Rightarrow \Diamond q) \wedge (q \Rightarrow \circ r) \wedge (r \Rightarrow \Diamond \Box s)) \Rightarrow (\Box p \Rightarrow \Diamond s)$

Proof:

$$
\begin{array}{ll}
 & \Box((p \Rightarrow \Diamond q) \wedge (q \Rightarrow \circ r) \wedge (r \Rightarrow \Diamond \Box s)) \Rightarrow (\Box p \Rightarrow \Diamond s) \\
= & \quad \langle \text{(99) Distributivity of } \Box \text{ over } \wedge \rangle \\
 & \Box(p \Rightarrow \Diamond q) \wedge \Box(q \Rightarrow \circ r) \wedge \Box(r \Rightarrow \Diamond \Box s) \Rightarrow (\Box p \Rightarrow \Diamond s) \\
= & \quad \langle \text{(3.65) Shunting, } p \wedge q \Rightarrow r \equiv p \Rightarrow (q \Rightarrow r) \rangle \\
 & \Box(p \Rightarrow \Diamond q) \wedge \Box(q \Rightarrow \circ r) \wedge \Box(r \Rightarrow \Diamond \Box s) \wedge \Box p \Rightarrow \Diamond s
\end{array}
$$

And now,

$$
\begin{array}{ll}
 & \Box(p \Rightarrow \Diamond q) \wedge \Box(q \Rightarrow \circ r) \wedge \Box(r \Rightarrow \Diamond \Box s) \wedge \Box p \\
\Rightarrow & \quad \langle \text{(47) Weakening of } \Diamond \text{ and (3.82a) Transitivity and} \\
 & \quad \text{(120) Monotonicity of } \Box \text{ and (4.3) Monotonicity of } \wedge \rangle \\
 & \Box(p \Rightarrow \Diamond q) \wedge \Box(q \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow \Diamond \Box s) \wedge \Box p \\
\Rightarrow & \quad \langle \text{(124) Catenation rule of } \Diamond \text{ and (4.3) Monotonicity of } \wedge \rangle \\
 & (p \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow \Diamond \Box s) \wedge \Box p \\
\Rightarrow & \quad \langle \text{(76) Strengthening of } \Box \text{ and (4.3) Monotonicity of } \wedge \rangle \\
 & (p \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow \Diamond \Box s) \wedge p \\
= & \quad \langle \text{(3.36) Symmetry of } \wedge, p \wedge q \equiv q \wedge p, \text{ twice} \rangle \\
 & p \wedge (p \Rightarrow \Diamond r) \wedge \Box(r \Rightarrow \Diamond \Box s) \\
\Rightarrow & \quad \langle \text{(3.77) Modus ponens, } p \wedge (p \Rightarrow q) \Rightarrow q \text{ and (4.3) Monotonicity of } \wedge \rangle \\
 & \Diamond r \wedge \Box(r \Rightarrow \Diamond \Box s) \\
\Rightarrow & \quad \langle \text{(119) Monotonicity of } \Diamond \text{ and (4.3) Monotonicity of } \wedge \rangle \\
 & \Diamond r \wedge (\Diamond r \Rightarrow \Diamond \Diamond \Box s) \\
= & \quad \langle \text{(50) Absorption of } \Diamond \rangle \\
 & \Diamond r \wedge (\Diamond r \Rightarrow \Diamond \Box s) \\
\Rightarrow & \quad \langle \text{(3.77) Modus ponens, } p \wedge (p \Rightarrow q) \Rightarrow q \rangle \\
 & \Diamond \Box s \\
\Rightarrow & \quad \langle \text{(77) Strengthening of } \Box \text{ and (119) Monotonicity of } \Diamond \\
 & \quad \text{and (50) Absorption of } \Diamond \rangle \\
 & \Diamond s \quad \blacksquare
\end{array}
$$

3 Lectures

Material for our in-class discussions and lectures will assist us in our cause: proving digital systems hardware and software correct.

The lectures are approximately in the order we will discuss them, but we will no doubt "jump" around as our class evolves. Remember this is a draft document that we are creating together as the course proceeds.

Additional on-line materials that will be very helpful to you getting the most out of this class have been recently created by two of the *old Jedi-Masters* of ACL2: Prof. J Moore and Prof. Warren A. Hunt, Jr. Drs. Moore and Hunt taught a short course on ACL2 during the period May 22-28, 2021 at the 10th Summer School on Formal Techniques. The material includes video recordings of many (but not all) of the lectures presented, as well as copies of the presentation slides and lecture notes. We will take on the task of connecting between this course and that information resource by referring to the Summer School materials in the appropriate places in this document.

To start this process we recommend you watch lecture 0 from the summer school after lecture 4 for cs340d. *Introduction to the ACL2 Part of the Summer School* (`https://youtu.be/xcXBOkp_w1s`).

This summer school video gives a quick overview of the ACL2 lectures at the 2021 Summer School on Formal Techniques.

3.1 Lecture 0 – Introduction course overview and fibonacci example

There is an official website for CS 340d at the homepage (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/cs340d.html`) where you can find this document, an html version of it, and any information concerning the conduct of this course. In particular, changes to due dates of homework or projects; meeting dates/times/location of office hours, classes or exams; and general announcements about the conduct of the course can be found.

The lectures for CS340d will roughly follow the topics outlined in the following two documents. The first document is a lightly revised version of J Moore's Recursion and Induction Notes available *here* (`https://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/papers-and-talks/class-lectures.pdf`).

And the second document is a set of slides for classroom presentations available *here* (`https://www.cs.utexas.edu/users/hunt/class/2021-fall/cs389r/lecture-6.pdf`).

We recommend you watch lecture 0 from the summer school now. *Introduction to the ACL2 Project* (`https://youtu.be/912RcvJrlk0`).

This summer school video provides a brief history of the development and application of ACL2 over a nearly 50 year period.

Lecture 0 (January 10, 2023) has now been completed. You can access the files presented during the lecture from the course homepage. at the homepage (`http://www.cs.utexas.edu/users/hunt/class/2023-spring/cs340d/cs340d.html`)

3.2 Lecture 1 – The course syllabus rules UT disclosures

Lecture 1 (January 12, 2023) has now been completed. Based on the survey taken as Quiz 0 we have added a Chapter to the CS340d on propositional logic called "Basic Logic Review". If you have any question on propositional logic and proving PC formula are theorems you can bring those questions to class or see any of the course staff in office hours. Prof. Hunt finished a review of course goals, syllabus and the state required UT disclosures (which can be found in the CS340d course book). The course then started with a review of functional programming (FP), building on the approach taken in the fibonacci lecture. Recall that the slides for the fibonacci lecture are posted on the class webpage.

3.3 Lecture 2 – Introduction to functional programming

Lecture 2 (January 17, 2023) has now been completed. See slides for lecture 2 posted on the class webpage. Lecture focus was on List Processing: definition of a proper list, recognizer function called true-listp, functions len, app, and rev.

3.4 Lecture 3 – Introduction to tracing and debugging

Lecture 3 (January 19, 2023) has now been completed. See slides for lecture 3 posted on the class webpage. Key, likely new, concepts include: TRACE$, using accumulators, tail recursive functions, functions for operating on trees and the flatten function.

3.5 Lecture 4 – Continue introduction to functional programming in ACL2

Lecture 4 (January 24, 2023) has now been completed. Focus for this lecture was set operations on list representations of a set. Defined a predicate recognizer of set elements (eqlablep x). Looked at the difference between sets and "bags": no-dups (duplicates). Set-union defined. Notice in the lecture how first functions are defined, and then properties of those functions are stated and then proven mathematically in the ACL2 logic. A big topic in this class is memory modeling. We can use lists as memory and define properties around look-up and writing to memory. This lecture also looked at associative memory. This can be modeled using assoc-list (or a-lists). There is likely to be a lot of memory modeling in future lectures, so make sure you get these early examples understood.

3.6 Lecture 5 – Build an expression evaluator

Lecture 5 (January 26, 2023) has now been completed. An efficient flatten function, called mc-flatten was defined. This function was invented by John McCarthy. Look him up, he was a very influential computer scientist and you should be informed of his contributions. A lot of this lecture was devoted to development of an expression evaluator. First we develop code that "recognizes" valid input to the evaluator. Next we develop the evaluator for this restricted expression language that evaluates correctly formed expressions and returns their value. This could be considered a small subset of the Lisp REPL (read-eval-print-loop)

3.7 Lecture 6 – ACL2 function definition

Lecture 6 (January 31, 2023) on deck. This lecture was canceled as the University was closed due to a winter storm. Changes to the C340d class syllabus will be made and posted on the class website and in this document.

Lecture will cover introduction to ACL2 "guards". Development of data and structure recognizers which will check for well-formed input to a function. Some examples using trees will be explored. Look for the connection between these recognizers and guards.

3.8 Lecture 7 – General correctness principles

Lecture 7 (February 2, 2023) on deck. This lecture was canceled as the University was closed due to a winter storm. Changes to the C340d class syllabus will be made and posted on the class website and in this document.

This lecture covers the principles and practical application of assertions, invariants in establishing the correctness of a program. This might also include the use of tracing, simulators, testing and setting breakpoints in the code.

3.9 Lecture 8 – Presentation and use of the ACL2 Logic

Lecture 8 (February 7, 2023) has now been completed. Over the next 3-4 weeks we will get into the logic of ACL2, and how this can be used to model and analyze software and hardware. A review of the expression evaluator that was presented in the last class was presented again. Key points of the presentation were:

- the exprp code is called a recognizer and determines whether or not the input to the function is well-formed, i.e. if the syntax of the expression is correct.
- the function evx (the evaluator) is guarded by exprp. This ensures that only valid (syntactically) expressions are presented for evaluation.
- A constant folding function (fc) for optimization of expressions was defined. A correctness property was defined for the behavior of fc, and proven correct.

```
(defthm fc-works-correctly
  (implies (and (exprp x)
                (integer-val-alistp a))
           (equal (evx (fc x) a)
                  (evx x a))))
```

The remainder of the lecture moved into topics that relate to the reading in the on-line documentation – Recursion and Induction. "Jump to" R-and-i-table-of contents to see the individual topics that are listed below. We have covered ACL2 data types (numbers, characters, strings, symbols and ordered pairs) in previous lectures. Now the key concepts to understand include:

- Terms – A term is a variable symbol, or a QUOTED constant, or a function application f of arity n, of n terms $(f\ x_1\ x_2\ \ldots\ x_n)$
- Abbreviations – "Jump to" R-and-i-abbreviations-for-terms to read the on-line documentation about abbreviations.

- Identity – the definition of identity is used to determine if two ACL2 objects are equivalent. If you write two objects in their canonical form, and those canonical forms are different, then the two objects are different. Each data type in ACL2 has a canonical form defined.
- Six primitive functions: (cons x y), (car x), (cdr x), (consp x), (if x y z), (equal x y).
- Substitutions – is a set sigma, of bindings of variables to terms { x <– (car a), y <– (cdr x) } Terms are substituted for variables in a term.

3.10 Lecture 9 –Terms and functions revisited

Lecture 9 (February 9, 2023) has now been completed. ACL2 Data types, working with terms, substitutions and Function definition revisited.

3.11 Lecture 10 – Terms and functions revisited

Lecture 10 (February 14, 2023) has now been completed. More on working with terms, substitutions and abbreviations and proof by case analysis. Function definition revisited and homework problem 12 done in class. ACL2 functions evaluate arguments left-to-right and are "call-by-value." Slide 15 in lecture notes shows how standard PC logical connectives (and, or, not, etc.) can all be written in terms of IF. Therefore the discussion in class of IF as the only logical connective necessary for writing PC formulas. Started a discussion of the components of a formal mathematical theory including: well-formed syntax for formulas, a set of axioms, and inference rules that derive new theorems from old ones. Next lecture will get into the details of the Axioms of ACL2, and begin looking at doing proofs by hand.

3.12 Lecture 11 – ACL2 revisited

Lecture 11 (February 16, 2023) has now been completed. Lab 0 discussion. Man page specifications can be different between different Unix or Linux OSs. The Lab 0 will be graded on its performance on UTCS linux machines. Some ACL2 lisp code was provided in the Laboratory assignment document. Unfortunately it was a pdf, and some students had trouble getting the code to run in ACL2. Prof. Hunt has now put the code up on the class website for easy access by all students. Any code provided for Lab 1 will also be found on the class website.

3.13 Lecture 12 – ACL2 Theory repeated

Lecture 12 (February 21, 2023) has now been completed. Prof. Hunt went back over many of the key theoretical topics of the ACL2 system from Axioms, terms, formulas and proof to the Definitional principle and Structural Induction. This covered roughly from axioms on page 19 of the class-lectures.pdf up to structural induction on page 43. Another view of this material can be found in the on-line documentation from section "r-and-i-axioms" to section r-and-i-structural-induction.

There was considerable time spent describing ACL2 use of terms in proofs instead of formulas. And it was pointed out that the homework assignments were not requiring students to do proofs of the equivalence of terms vs formulas as in problems 18 - 26.

The student should nevertheless feel free to work out proofs of the problems 18 - 26 as time permits. To help such an effort, we provide a proof of Problem 23 and Problem 20 as an example of how to proceed.

Some Example Proofs

Solutions for some of the proofs in problems 18-23 are presented as extra examples of proving theorems in the propositional logic. Lemma A is repeated here for easy reference. Numbers like (3.14) below reference theorems from my favorite equations sheet which is available on the class website. The file is "vega-equations-new.pdf" and it may be helpful in constructing PC proofs.

Lemma A: $(p \not\equiv nil) \equiv p$
Proof: Start with LHS and show LHS ≡ RHS.

$(p \not\equiv nil)$

$\equiv \langle$(3.14) $(p \not\equiv nil) \equiv p$ with q := $nil\rangle$

$\neg p \equiv nil$

$\equiv \langle$ (3.2) Symmetry of $\equiv$ and (3.15) $\neg p \equiv p \equiv nil\rangle$

p

QED

Problem 23. Prove the following is a theorem.
(implies (and p (implies q r)) s) $\equiv (p \wedge (q \Rightarrow r)) \Rightarrow s)$

Proof: Start with LHS and show LHS ≡ RHS.

(implies (and p (implies q r)) s)

$\equiv \langle$ Problem 20 and Lemma A with $p := q, q := r\rangle$

(implies (and p ($q \Rightarrow r$))s)

$\equiv \langle$ Problem 18 and Lemma A with $q := (q \Rightarrow r)\rangle$

(implies (p $\wedge(q \Rightarrow r)$)s)

$\equiv \langle$ Problem 20 and Lemma A with $p := p \wedge (q \Rightarrow r), q := s\rangle$

$(p \wedge (q \Rightarrow r)) \Rightarrow s)$

QED

Problem 20. Prove the following is a theorem.
(implies p q) $\neq nil \equiv (p \neq nil) \Rightarrow (q \neq nil)$

Proof: By (4.7) Mutual Implication.
Proof in the first direction follows. Use (3.65) shunting to start.

((implies p q) $\neq nil) \wedge (p \neq nil) \Rightarrow (q \neq nil)$

Start with LHS and show it implies the RHS.
((implies p q) $\neq nil) \wedge (p \neq nil)$

$\equiv \langle$ Definition of implies $\rangle$

(if p (if q t nil) t) $\neq nil \wedge (p \neq nil)$

$\equiv \langle$ Semantics of if with $(p \neq nil)\rangle$

(if q t nil) $\neq nil$

$\equiv \langle$ Semantics of if $\rangle$

$q \neq nil$

QED

Proof in the second direction follows. Use (3.65) shunting again, to get
$(p \neq nil) \wedge (q \neq nil) \Rightarrow$ (implies p q) $\neq nil)$
By (4.4) Deduction, assume $(p \neq nil), (q \neq nil)$
and prove
(implies p q)$\neq nil$

$\equiv \langle$ Definition of implies $\rangle$

(if p (if q t nil) t) $\neq nil)$

$\equiv \langle$ Assumed conjuncts of antecedent: $(p \neq nil), (q \neq nil)\rangle$

$t \neq nil$ — (Axiom 1)

QED

3.14 Lecture 13 – ACL2 Axioms

Lecture 13 (February 23, 2023) has now been completed. Key topics covered include: the definitional principle (simpliifed). The measure variable and the car/cdr nest. Structural induction as the key proof method.

3.15 Lecture 14 – Proof by Induction

Lecture 14 (February 28, 2023) has now been completed. The focus of this lecture and indeed the attention of the whole course is directed on computation by recursion and proof of properties of this computation by induction.

```
Given the function (f x) defined as follows:

(defun f (x)
  (if (consp x)
      (f (cdr x))
      t))

Does this function meet the criteria in the Definitional Principle?
Let's check.

      1. The function symbol being defined is new. Yes.
      2. The formal variables are distinct. There is only one formal variable, x.
      3. The body is a term, and the only variable symbol, x, is a formal.
      4. There is a ruler for the recursive call of F, (consp x). And, there is
          a car/cdr nest around them measured formal, x.

So, Yes, we will admit this function definition into the logic.

And for the final check. ACL2 is always the final check on admissibility. Here
is what ACL2 has to say.

ACL2 !>  (DEFUN F (X)
                  (IF (CONSP X)
                      (F (CDR X))
                      T))

The admission of F is trivial, using the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT X).  We observe that the type of F is described by the
theorem (EQUAL (F X) T).
```

```
Summary
Form:  ( DEFUN F ...)
Rules: NIL
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 F
ACL2 !>

Now we will prove: (equal (f x) t)

Perhaps we can prove it by induction.

As X is the only variable, we will induct on X.

Base Case:
(implies (not (consp x))   ;; Hyp1 - assume (not (consp x)) is true
              (equal (f x) t))

< Def. of (f x) >

(implies (not (consp x))   ;; Hyp1
              (equal (if (consp x)
                         (f (cdr x)) t) t))

< Hyp1 >

(implies (not (consp x))   ;; Hyp1
              (equal (if NIL
                         (f (cdr x)) t) t))

< Axiom 3 >
(implies (not (consp x))   ;; Hyp1
              (equal t t))

This proves the Base case.
```

QED

```
Induction Substitution:  (x (cdr x))

Induction Step:


(implies (and (consp x)                ;; hyp1
              (equal (f (cdr x)) t))   ;; IH1
         (equal (f x) t))
```

```
< Def. (f x) >

(implies (and (consp x)                ;; hyp1
              (equal (f (cdr x)) t))   ;; IH1
          (equal (if (consp x)
                     (f (cdr x)) t) t))

< hyp1 >

(implies (and (consp x)                 ;; hyp1
              (equal (f (cdr x)) t))    ;; IH1
         (equal (f (cdr x)) t))

But this is just the Induction Hypothesis. Therefore this
reduces to

< IH1 >

(implies (and (consp x)                 ;; hyp1
              (equal (f (cdr x)) t))    ;; IH1
          t)
```

QED

3.16 Lecture 15 – Assoc of App

Lecture 15 (March 2, 2023) has now been completed. This lecture presented a detailed walk-through of a hand proof of associativity of Append, shown below. To check your understanding of the the logical steps of the proof, supply the justification for each step of the proof below where you find a ** hint ** placeholder.

Proof of Associativity of Append

```
< Definition (Def.) of ap >

(ap x y)
 =
(if (consp x)
    (cons (car x)
          (ap (cdr x) y))
  y)

< Prove the following conjecture >

Conjecture:  (equal (ap (ap x y) z)
                    (ap x (ap y z)))
```

```
Choose induction variable:  X, and the
car/cdr substitution (sigma):  ((x (cdr x))
                                (y y)
                                (z z))

Base case:

(implies (not (consp x))                ; hyp1
         (equal (ap (ap x y) z)         ; (equal LHS
                (ap x (ap y z))))        ;       RHS)

=  < Def. ap  - second ap on LHS >

(implies (not (consp x))
         (equal (ap (if (consp x)
                        (cons (car x)
                              (ap (cdr x) y))
                      y) z)
                (ap x (ap y z))))

=  < hyp1 (consp x) = NIL >

(implies (not (consp x))
         (equal (ap (if NIL
                        (cons (car x)
                              (ap (cdr x) y))
                      y) z)
                (ap x (ap y z))))

=  <  ** hint ** >


(implies (not (consp x))
         (equal (ap y z)
                (ap x (ap y z))))

=  <  ** hint ** >


(implies (not (consp x))
         (equal (ap y z)
                (if (consp x)
                    (cons (car x)
                          (ap (cdr x) (ap y z)))
                  (ap y z))))

=  < hyp1 (consp x) = NIL >
```

```
(implies (not (consp x))
         (equal (ap y z)
                (if NIL
                    (cons (car x)
                          (ap (cdr x) (ap y z)))
                  (ap y z))))

=  <  ** hint ** >

(implies (not (consp x))
         (equal (ap y z)
                (ap y z)))
```

QED

This completes the proof of the Base Case. Now for proof of the Induction Step.

```
Induction Step:

car/cdr substitution (sigma):  ((x (cdr x))
                                (y y)
                                (z z))

(implies (and (consp x)
              {(equal (ap (ap x y) z)
                      (ap x (ap y z)))/sigma}
              )
         (equal (ap (ap x y) z)
                (ap x (ap y z))))

Second part of proof by Induction:  Induction Step:

(implies (and (consp x)                              ; hyp1
              (equal (ap (ap (cdr x) y) z)           ; hyp2
                     (ap (cdr x) (ap y z))))
         (equal (ap (ap x y) z)                     ; (equal LHS
                (ap x (ap y z))))                   ;        RHS)

<  Same goal with different notation  >

(implies (and (consp x)
              (equal (ap (ap (cdr x) y) z)
                     (ap (cdr x) (ap y z))))
         (equal (ap (ap x y) z)                     ;  1
                (ap x (ap y z))))                   ;  2
```

```
<  ;1  Start with LHS  >
(ap (ap x y) z)

=  < Def. ap - second ap on LHS >

(ap (if (consp x)
        (cons (car x)
              (ap (cdr x) y))
      y)
    z)

=  <  ** hint ** >

(ap (if T
        (cons (car x)
              (ap (cdr x) y))
      y)
    z)

=  <  ** hint ** >

(ap (cons (car x)
          (ap (cdr x) y))
    z)

=  <  Def. of ap with the following substitutions
        x = (cons (car x) (ap (cdr x) y)) and y = z >

(if (consp (cons (car x) (ap (cdr x) y)))
    (cons (car (cons (car x) (ap (cdr x) y)))
          (ap (cdr (cons (car x) (ap (cdr x) y))) z))
  z)

=  <  ** hint ** >

(if T
    (cons (car (cons (car x) (ap (cdr x) y)))
          (ap (cdr (cons (car x) (ap (cdr x) y))) z))
  z)

=  <  ** hint ** >

(cons (car (cons (car x) (ap (cdr x) y)))
          (ap (cdr (cons (car x) (ap (cdr x) y))) z))

=  <  ** hint ** >
```

```
(cons (car x)
          (ap (cdr (cons (car x) (ap (cdr x) y))) z))

=  <  ** hint ** >


(cons (car x)
          (ap (ap (cdr x) y) z))

=  < hyp2 - Induction Hypothesis  >

(cons (car x)
          (ap (cdr x) (ap y z)))


< We hold here with our work on the LHS, and work with the RHS of the goal >


<  ;2  Start with RHS  >
(ap x (ap y z))

=  <  Def. of ap with the following substitution x = x and y = (ap y z) >

(if (consp x)
    (cons (car x)
          (ap (cdr x) (ap y z)))
  (ap y z))

=  < hyp1 (consp x) = T >

(if T
    (cons (car x)
          (ap (cdr x) (ap y z)))
  (ap y z))

=  <  ** hint ** >


(cons (car x)
          (ap (cdr x) (ap y z)))


< We hold here with our work on the RHS, and consider the state of the proof >

We have now shown on track ;1 that

(equal (ap (ap x y) z)
```

```
        (cons (car x) (ap (cdr x) (ap y z))))          ; Is a theorem

And we have shown on track ;2 that

(equal (ap x (ap y z))
        (cons (car x) (ap (cdr x) (ap y z))))           ; Is a theorem

So by the propositional calculus theorem, (3.51) Replacement we can say
that the

Conjecture:  (equal (ap (ap x y) z)
                    (ap x (ap y z)))

holds. Therefore having completed the required proofs, our qoal
formula (conjecture)  holds under the principle of structural induction.
```

QED

3.17 Lecture 16 – Storing values in variables

Lecture 16 (March 7, 2023) has now been completed. Prof. Hunt discussed the debug helper macros (! x y), and (!! x y) that were included in code for working on the CS340d Labs. These two macros provide a way to mimic storing global variables. For example, the command (! x 10) works roughly like the Common Lisp expression (setq x 10). In ACL2, the value stored in x is retrieved using the macro @ as follows. (@ x).

```
(defmacro ! (x y)
  (declare (xargs :guard (symbolp x)))
  `(assign ,x ,y))

(defmacro !! (variable new-value)
  ;; Assign without printing the result.
  (declare (xargs :guard t))
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (if erp 'Error! ',variable))))

The macro @ gives convenient access to the value of such globals.
 For example:
 ACL2 >(! x 7)
 7
ACL2 >(! y 10)
 10
ACL2 >(+ (@ x) (@ y))
```

17

The use of *defthm* as a debugging and verifying tool was presented. Defthm is used to prove properties of functions (which are components) of the solution you are developing. This allows building and incrementally checking that parts of your system's design are correct. You will often see the following pattern in the modeling of systems using ACL2.

```
(defun f1 (x y ...))
(defthm property-1-of-f1  ...)    ;; Prove f1 has this property
(defthm property-2-of-f1  ...)
.
.
.
(defthm property-x-of-f1  ...)

(defun f2 (x y ...))
(defthm property-1-of-f2  ...)    ;; Prove f2 has this property
(defthm property-2-of-f2  ...)
.
.
.
(defthm property-x-of-f2  ...)
```

```
Here is a concrete example from Prof. Hunt's Lab1 code.
Don't worry about the details, just notice the pattern above
in the code.
```

```
(defun advance-to-space-and-collect (char-lst accum-list)
  (declare (xargs :guard (and (character-listp char-lst)
                              (character-listp accum-list))))
  (if (atom char-lst)
      (mv char-lst (reverse accum-list))
    (let ((char (car char-lst)))
      (if (member char *space-seperators*)
          (mv char-lst (reverse accum-list))
        (advance-to-space-and-collect (cdr char-lst)
                                      (cons char accum-list))))))

(defthm character-listp-car--advance-to-space-and-collect
  (implies (and (character-listp char-lst)
                (character-listp accum-list))
           (character-listp (car (advance-to-space-and-collect
                                  char-lst accum-list)))))

(defthm character-listp-cadr-advance-to-space-and-collect
  (implies (and (character-listp char-lst)
```

```
              (character-listp accum-list))
         (character-listp (cadr (advance-to-space-and-collect
                                 char-lst accum-list)))))

(defthm len-char-lst-after-advance-to-space-and-collect-<=
  (<= (len (car (advance-to-space-and-collect char-lst accum-list)))
      (len char-lst))
  :rule-classes (:linear :rewrite))
```

The rest of the lecture time was spent introducing Lab2 and discussing the list set recognizer *setp* and a binary tree recognizer *bstp*.

3.18 Lecture 17 – Problem 43 and Proof process

Lecture 17 (March 9, 2023) has now been completed. Problem 43 and Proof process. There are many ways any proof challenge can be met. There is no one way to approach it. In Problem 43, from the homework, if you go straight to opening up function definitions and trying to simplify the resulting terms, you may be dealing with a lot of complexity. Nevertheless, we can prove the base case in the usual way.

```
Problem 43.
Prove

(equal (rev (mapnil x)) (mapnil (rev x)))      ;(equal LHS RHS)

; -------------------------------------------------------------
; Append function

(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

; -------------------------------------------------------------
; Reverse function

(defun rev (x)
  (if (consp x)
      (app (rev (cdr x))
           (cons (car x) nil))
    nil))

; -------------------------------------------------------------
; mapnil function

(DEFUN MAPNIL (X)
  (IF (CONSP X)
      (CONS NIL (MAPNIL (CDR X)))
```

```
      NIL))

;;
;; Some Lemmas that might be helpful, or not.

MAPNIL-APP

(DEFTHM MAPNIL-APP
  (EQUAL (MAPNIL (APP A B))
         (APP (MAPNIL A) (MAPNIL B))))

MAPNIL-CDR

(DEFTHM MAPNIL-CDR
  (EQUAL (MAPNIL (CDR X))
         (CDR (MAPNIL X))))

APP-NIL

(defthm app-nil
  (implies (true-listp x)
           (equal (app x nil)
                  x)))
APP-REV

(defthm app-rev
  (equal (app (rev x) nil)
         (rev x)))

REV-APP

(defthm rev-app
  (equal (rev (app a b))
         (app (rev b) (rev a))))
  ;;
  ;; Axioms created by the Definitional Principle.
  ;;

  < Def. App >

  (app x y)
  =
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
    y)
```

```
  < Def. rev >

  (rev x)
  =
  (if (consp x)
      (app (rev (cdr x))
           (cons (car x) nil))
    nil)

  (MAPNIL X)
  =
  (IF (CONSP X)
      (CONS NIL (MAPNIL (CDR X)))
      NIL))


< Perhaps we can prove it by induction. >

Base Case:

(implies (not (consp x))
         (equal (rev (mapnil x)) (mapnil (rev x))))      ;(equal LHS RHS)

=  < Def. mapnil >

(implies (not (consp x))             ;; hyp1
         (equal (rev (IF (CONSP X)
                         (CONS NIL (MAPNIL (CDR X)))
                         NIL))
                (mapnil (rev x))))

=  < hyp1 and Axiom 3 >

(implies (not (consp x))             ;; hyp1
         (equal (rev NIL)
                (mapnil (rev x))))

=  < Def. rev >

(implies (not (consp x))             ;; hyp1
         (equal (rev NIL)
                (mapnil (if (consp x)
                            (app (rev (cdr x))
                                 (cons (car x) nil))
                          nil))))

=  < hyp1 and Axiom 3 >
```

```
(implies (not (consp x))              ;; hyp1
(equal (rev NIL)
       (mapnil NIL)))

=  < Def. rev and Def. mapnil  >

(implies (not (consp x))              ;; hyp1
  (equal NIL NIL))

T
```

QED

`This proves the Base Case.`

Now, normally we would formulate the induction step with an induction variable and a car/cdr substitution to set up the induction hypothesis. However, in the proof provided by Prof. Hunt he shows two lemmas that significantly simplifies the proof.

`The first lemma is:`

```
(equal (rev (mapnil x))
        (mapnil x))
```

`The second lemma is:`

```
(equal (mapnil (rev x))
       (mapnil x))
```

`This makes the proof of the Main Result trivial.`

```
(equal (rev (mapnil x))
       (mapnil (rev x)))

= < Lemma 1 and Lemma 2 >

(equal (mapnil x)
       (mapnil x))

= < Reflexivity of = >

T
```

QED

The last topic in this lecture was further discussion of Lab2. The six required functions to be written for the lab were explained, along with the recognizers *setp* and *bstp*. Prof. Hunt also showed how to write the *insrt* function. It was a Spring Break present.

```
(defun insrt (e X)
  "Insert e into ordered set X."
  (declare (xargs :guard (and (atom e)
                              (setp X))))
  ;; Replace X (below) with an Insert function body
  (if (atom X)
      (list e)
      (if (<< e (car X))
          (cons e X)
          (cons (car X)
                (insrt e (cdr X))))))
```

3.19 Lecture 18 – Verification of iSort

Lecture 18 (March 21, 2023) has now been completed. The slides for the presentation "Verification of iSort" are available on the course webpage. Prof. Hunt spent time showing the class how to approach the proof of Problem 61 (equal (rev2 x nil) (rev x)). See also Lecture 21 for more on Problem 61.

3.20 Lecture 19 – Array-based iSort

Lecture 19 (March 23, 2023) has now been completed. Continued with the iSort talk, extending it to in-place sorting. In the talk memory is modeled as a list of integers of length n accessed through a natural number index 0 - n-1. Slides for the lecture are posted on the course web site.

3.21 Lecture 20 – The Method

Lecture 20 (March 28, 2023) has now been completed. Maxine presented a review of the last quiz. There was an argument of an IF function that was missing which made some questions of the quiz ambiguous. The grading of the quiz will take this confusion into consideration. The complete proof from the quiz is included under Quiz 14 section of the class notes.

Prof. Hunt began discussion on "The Method." Your should read about The Method in the online documentation. You can get to it by typing "The-Method" into the "Jump to" box.

You can also check out the "Proof-builder" in the online documentation. To get you started, here is a short list of the most commonly used Proof-builder commands.

```
ACL2-pc::=
    (atomic macro) attempt an equality (or equivalence) substitution
ACL2-pc::bash
    (atomic macro) call the ACL2 theorem prover's simplifier
ACL2-pc::bk
    (atomic macro) move backward one argument in the enclosing term
ACL2-pc::cg
```

```
    (macro) change to another goal.
ACL2-pc::claim
    (atomic macro) add a new hypothesis
ACL2-pc::comm
    (macro) display instructions from the current interactive session
ACL2-pc::contrapose
    (primitive) switch a hypothesis with the conclusion, negating both
ACL2-pc::demote
    (primitive) move top-level hypotheses to the conclusion
ACL2-pc::drop
    (primitive) drop top-level hypotheses
ACL2-pc::dv
    (atomic macro) move to the indicated subterm
ACL2-pc::exit
    (meta) exit the interactive proof-builder
ACL2-pc::expand
    (primitive) expand the current function call without simplification
ACL2-pc::goals
    (macro) list the names of goals on the stack
ACL2-pc::in-theory
    (primitive) set the current proof-builder theory
ACL2-pc::induct
    (atomic macro) generate subgoals using induction
ACL2-pc::nx
    (atomic macro) move forward one argument in the enclosing term
ACL2-pc::p
    (macro) prettyprint the current term in the usual user-level (untranslated) syntax
ACL2-pc::p-top
    (macro) prettyprint the conclusion, highlighting the current term
ACL2-pc::prove
    (primitive) call the ACL2 theorem prover to prove the current goal
ACL2-pc::r
    (macro) same as rewrite
ACL2-pc::replay
    (macro) replay one or more instructions
ACL2-pc::restore
    (meta) remove the effect of an UNDO command
ACL2-pc::retrieve
    (macro) re-enter the proof-builder
ACL2-pc::runes
    (macro) print the runes (definitions, lemmas, ...) used
ACL2-pc::s
    (primitive) simplify the current subterm
ACL2-pc::s-prop
    (atomic macro) simplify propositionally
ACL2-pc::save
    (macro) save the proof-builder state (state-stack)
```

```
ACL2-pc::show-rewrites
    (macro) display the applicable rewrite rules
ACL2-pc::split
    (atomic macro) split the current goal into cases
ACL2-pc::sr
    (macro) same as SHOW-REWRITES
ACL2-pc::th
    (macro) print the top-level hypotheses and the current subterm
ACL2-pc::top
    (atomic macro) move to the top of the goal
ACL2-pc::undo
    (meta) undo some instructions
ACL2-pc::up
    (primitive) move to the parent (or some ancestor) of the current subterm
ACL2-pc::use
    (atomic macro) use a lemma instance
ACL2-pc::x
    (atomic macro) expand and (maybe) simplify function call at the current subterm
ACL2-pc::x-dumb
    (atomic macro) expand function call at the current subterm, without simplifying
```

3.22 Lecture 21 – Proof Automation

Lecture 21 (March 30, 2023) has now been completed.

Class TA reviewed one of our quiz problems. This can be found under the section entitled "Quiz 14" in the Class Book. We discussed the ACL2 proof method and Prof. Hunt demonstrated this method working on Prob. 61 (one of the homework problems).

```
;;;
;;;
;;; Problem 61. Prove (equal (rev1 x nil)(rev x))

;;
;; rev1 - tail recursive reverse
;;
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
    a))

;;
;; app - append
;;
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))
```

```
;;
;; rev - reverse
;;
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x))
           (cons (car x) nil))
    nil))

;;
;; How to find this key-lemma using
;; ``The Method'' is the art of ACL2
;;
(defthm key-lemma
  (equal (rev1 x y)
         (app (rev x) y)))

;;
;; The Main Result
;;
(defthm rev1-rev-equiv
  (equal (rev1 x nil)
         (rev x)))
```

The above 5 events product the following proof description output.

```
The admission of REV1 is trivial, using the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT X).  We observe that the type of REV1 is described by the
theorem (OR (CONSP (REV1 X A)) (EQUAL (REV1 X A) A)).  We used primitive
type reasoning.

Summary
Form:  ( DEFUN REV1 ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 REV1
ACL2 !>
The admission of APP is trivial, using the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT X).  We observe that the type of APP is described by the
theorem (OR (CONSP (APP X Y)) (EQUAL (APP X Y) Y)).  We used primitive
type reasoning.

Summary
```

```
Form:  ( DEFUN APP ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 APP
ACL2 !>
The admission of REV is trivial, using the relation O< (which is known
to be well-founded on the domain recognized by O-P) and the measure
(ACL2-COUNT X).  We observe that the type of REV is described by the
theorem (OR (CONSP (REV X)) (EQUAL (REV X) NIL)).  We used primitive
type reasoning and the :type-prescription rule APP.

Summary
Form:  ( DEFUN REV ...)
Rules: ((:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:TYPE-PRESCRIPTION APP))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 REV
ACL2 !>
*1 (the initial Goal, a key checkpoint) is pushed for proof by induction.

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  Subsumption reduces that number to one.

We will induct according to a scheme suggested by (REV1 X Y).  This
suggestion was produced using the :induction rules REV and REV1.  If
we let (:P X Y) denote *1 above then the induction scheme we'll use
is
(AND (IMPLIES (NOT (CONSP X)) (:P X Y))
     (IMPLIES (AND (CONSP X)
                   (:P (CDR X) (CONS (CAR X) Y)))
              (:P X Y))).
This induction is justified by the same argument used to admit REV1.
Note, however, that the unmeasured variable Y is being instantiated.
When applied to the goal at hand the above induction scheme produces
two nontautological subgoals.
Subgoal *1/2
Subgoal *1/1
Subgoal *1/1'
Subgoal *1/1''
Subgoal *1/1'''
Subgoal *1/1'4'
Subgoal *1/1'5'

([ A key checkpoint while proving *1 (descended from Goal):

Subgoal *1/1'
(IMPLIES (AND (CONSP X)
```

```
              (EQUAL (REV1 (CDR X) (CONS (CAR X) Y))
                     (APP (REV (CDR X)) (CONS (CAR X) Y))))
          (EQUAL (REV1 (CDR X) (CONS (CAR X) Y))
                 (APP (APP (REV (CDR X)) (LIST (CAR X)))
                      Y)))

*1.1 (Subgoal *1/1'5') is pushed for proof by induction.

])

So we now return to *1.1, which is

(EQUAL (APP RV (CONS X1 Y))
       (APP (APP RV (LIST X1)) Y)).
Subgoal *1.1/2
Subgoal *1.1/1

*1.1 and *1 are COMPLETED!
Thus key checkpoints Subgoal *1/1' and Goal are COMPLETED!

Q.E.D.

Summary
Form:  ( DEFTHM KEY-LEMMA ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION REV)
        (:DEFINITION REV1)
        (:ELIM CAR-CDR-ELIM)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:INDUCTION REV)
        (:INDUCTION REV1)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS))
Time:  0.01 seconds (prove: 0.01, print: 0.00, other: 0.00)
Prover steps counted:  1419
 KEY-LEMMA
ACL2 !>
ACL2 Warning [Subsume] in ( DEFTHM REV1-REV-EQUIV ...):  The previously
added rule KEY-LEMMA subsumes a newly proposed :REWRITE rule generated
from REV1-REV-EQUIV, in the sense that the old rule rewrites a more
general target.  Because the new rule will be tried first, it may nonetheless
find application.

Goal'
```

```
([ A key checkpoint:

Goal'
(EQUAL (APP (REV X) NIL) (REV X))

*1 (Goal') is pushed for proof by induction.

])

Perhaps we can prove *1 by induction.  Two induction schemes are suggested
by this conjecture.  Subsumption reduces that number to one.

We will induct according to a scheme suggested by (REV X).  This suggestion
was produced using the :induction rule REV.  If we let (:P X) denote
*1 above then the induction scheme we'll use is
(AND (IMPLIES (NOT (CONSP X)) (:P X))
     (IMPLIES (AND (CONSP X) (:P (CDR X)))
              (:P X))).
This induction is justified by the same argument used to admit REV.
When applied to the goal at hand the above induction scheme produces
two nontautological subgoals.
Subgoal *1/2
Subgoal *1/1
Subgoal *1/1'
Subgoal *1/1''
Subgoal *1/1'''
Subgoal *1/1'4'
Subgoal *1/1'5'

([ A key checkpoint while proving *1 (descended from Goal'):

Subgoal *1/1'
(IMPLIES (AND (CONSP X)
              (EQUAL (APP (REV (CDR X)) NIL)
                     (REV (CDR X))))
         (EQUAL (APP (APP (REV (CDR X)) (LIST (CAR X)))
                     NIL)
                (APP (REV (CDR X)) (LIST (CAR X))))).

*1.1 (Subgoal *1/1'5') is pushed for proof by induction.

])

So we now return to *1.1, which is

(EQUAL (APP (APP RV (LIST X1)) NIL)
       (APP (APP RV NIL) (LIST X1))).
```

```
Subgoal *1.1/2
Subgoal *1.1/1

*1.1 and *1 are COMPLETED!
Thus key checkpoints Subgoal *1/1' and Goal' are COMPLETED!

Q.E.D.

Summary
Form:  ( DEFTHM REV1-REV-EQUIV ...)
Rules: ((:DEFINITION APP)
        (:DEFINITION REV)
        (:ELIM CAR-CDR-ELIM)
        (:EXECUTABLE-COUNTERPART APP)
        (:EXECUTABLE-COUNTERPART CONSP)
        (:EXECUTABLE-COUNTERPART EQUAL)
        (:FAKE-RUNE-FOR-TYPE-SET NIL)
        (:INDUCTION APP)
        (:INDUCTION REV)
        (:REWRITE CAR-CONS)
        (:REWRITE CDR-CONS)
        (:REWRITE KEY-LEMMA))
Warnings:  Subsume
Time:  0.01 seconds (prove: 0.01, print: 0.00, other: 0.00)
Prover steps counted:  1572
 REV1-REV-EQUIV
ACL2 !>
```

3.23 Lecture 22 – The Method

Lecture 22 (April 4, 2023) has now been completed. Presentation of Lab 3 requirements, code templates, and timing.

3.24 Lecture 23 – Peano Arithmetic

Lecture 23 (April 6, 2023) has now been completed.

Discussed questions on the class piazza site about implementing the bst-del function. After a delete operation, the tree must still be ordered. Next was a discussion of Peano arithmetic, and how to implement arithmetic in ACL2. Defined a number of ACL2 functions to create natural numbers as the length of certain lists. Then defined the operations of plus and times to operate on these lists. Here is where we got with this exercise. Problem 67 is the times function, which is part of homework 10 (Problems 67 - 73).

```
;;;
;;;
;;; Peano Arithmetic
```

```
;;
;; Here is how we will define natural numbers.
(defun nat (x)
  (if (consp x)
      (and (equal (car x) nil)
           (nat (cdr x)))
      (equal x nil)))

(defthm true-listp-nat
    (implies (nat x)
             (true-listp x)))

;;
;; app - append
;;
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
      y))

;;
;;
(defun mapnil (x)
  (if (consp x)
      (cons nil (mapnil (cdr x)))
      nil))

(defthm true-listp-mapnil
    (true-listp (mapnil x)))

;;
;; Here is the additon operation.
(defun plus (x y)
  (if (atom x)
      (mapnil y)
      (cons nil (plus (cdr x) y))))

(defthm len-mapnil
    (equal (len (mapnil x))
           (len x)))

(defthm len-plus
    (equal (len (plus x y))
           (+ (len x) (len y))))

;;
```

```
;; Here is multiplication.
(defun times (x y)
  (if (atom x)
      nil
      (plus y (times (cdr x) y))))

(defthm true-listp-times
    (true-listp (times x y)))

(defthm times-is-commutative
    (implies (and (nat x) (nat y))
             (equal (times x y) (times y x))))

(defthm times-x-0
    (implies (atom y)
             (equal (times x y)
                    nil)))

;;
;; Here is an optimized multiplication,
;; suggested by a classmate.
(defun times-opt (x y)
  (if (or (atom x) (atom y))
      nil
      (times x y)))

(defthm times-opt-equiv-times
    (equal (times-opt x y)
           (times x y)))

;;
;; Here is a multiplication that uses
;; the best definition for proving theorems,
;; and the best one for efficient execution
;; as a simulator.
(defun uber-times (x y)
  (mbe :logic (times x y)
       :exec  (times-opt x y)))

(defthm true-listp-uber-times
    (true-listp (uber-times x y)))


(defthm uber-times-equiv-times-opt
    (equal (uber-times x y)
           (times-opt x y)))
```

3.25 Lecture 24 – Structural Induction

Lecture 24 (April 11, 2023) has now been completed.

Prof. Hunt added a file to the class web-page called sets-definitions.lisp You can find a link to the file under the links to Lab-0 through Lab-3 on the web-page. The lecture focused on going through this file in some detail. This thorough recap of Lab-2 functions, and more, should help with the work students are currently engaged in on Lab-3. You should feel free to use the functions in the sets-definitions.lisp file as the basis for working the proofs required for Lab-3.

3.26 Lecture 25 – popcount

Lecture 25 (April 13, 2023) has now been completed.

This lecture looked at aspects of the Peano arithmetic that the students are working on for homework #10. The principle of structural recursion is extended beyond our current definition that requires a car/cdr nest to guarantees that something gets smaller everytime through the iteration so that it can be proved that the function terminates.

Prof. Hunt finished the class with an example of using ACL2 to model a C program called popcount. Popcount counts the number of 1s in a computer word (32 or 64 bit). The approach was to create a shallow-embedding of the C program into acl2. This was done even though C does not have a formal semantics. The Popcount talk will be posted on the class website.

3.27 Lecture 26 – Verification and Validation

Lecture 26 (April 18, 2023) has now been completed. This class had two parts. In the first part, Prof. Hunt answered questions that students had on proving the theorems required to complete Lab-3. In particular, there were some helper lemmas needed to prove some of the theorems. Examples of how to identify the needed lemmas and prove them were given.

In the second part of the class, Scott presented some thoughts on how this class, CS340d, fits into the larger topic of program verification and validation (V & V). Program V & V is a field of its own with a big literature, textbooks, and even an IEEE standard for writing a software requirements document. Some of the material that was discussed is available on the class website.

3.28 Lecture 27 – The Last Class

Lecture 27 (April 20, 2023) has now been completed. This is the last class. Congratulations on the new skills you have acquired by staying with the significant amount of work required to complete the course. With your increased level of understanding of how to prove theorems in ACL2, it is our recommendation that you now go back to where most people start their learning of ACL2. Look at the **frequently-asked-questions-by-newcomers** and the tutorial **introduction-to-the-theorem-prover**, both in the ACL2 online manual.

Prof. Hunt presented a collection of more advanced topics that ACL2 has for controlling at a more granular level. This material came from a presentation that J Moore gave to the Stanford Summer School that has been referenced many times in the Class Book.

Next, a 20 minute quiz was given. This was followed by a presentation on using ACL2 in modeling and eventually building computer hardware. Many different types of hardware

have been modeled and proven to implement their specification. Some of this work goes on daily in industry. You can conduct your own search and investigation to see the extent of the use of this technology in application today.

4 CS340d Quizzes

Quizzes are designed to give us (and, you) a handle on your understanding of the various concepts related to this course. Quizzes may be given at any time, and there may be multiple quizzes in a class. Quizzes may involve writing some code so it is important that you bring you laptop to every class. Quizzes also give the students insight into what types of questions and problems are likely to be on exams.

4.1 Quiz 0 Welcome Questionnaire

```
Quiz 0
CS 340d
Unique Number: 52285
Spring, 2023

Given: January 10, 2023
```

Welcome to CS340d Debugging and Verifying Programs Spring 2023

Welcome

Welcome to CS340d: Debugging and Verifying Programs. We are glad you are joining us this semester. In recent semesters past it has been a challenge to navigate life under the COVID-19 pandemic. Particularly as it pertains to providing a world-class educational experience. But, we have learned a lot through this challenge and are committed to applying this experience working with you to achieve your learning goals at UTA. To start with, we would like to get to know you a little better. Please fill out the following short questionnaire which will give us some data and some feelings about where you are at the start of this semester and course. WAHJr, VR, SMS.

OUR GET TO KNOW YOU QUESTIONNAIRE

```
Contact Information

        Name:

        Preferred Name (What do your friends call you?):

        Contact information: (How can we contact you and get information
                             to you, during the semester e.g. e-mail,
                             mobile phone, fax, USPS mail?):
        Pronouns:

        A reminder: Please take the time to update your contact information in
                    CANVAS, Piazza, and Zoom.
```

This can come in handy as the semester proceeds.

Academic Background

What are your favorite CS topics (e.g., Programming Languages, Operating Systems, Databases, AI and ML, Algorithms, etc.)?

What CS, and other courses (math, science, engineering, etc.) have you taken that you feel prepare you to take CS340d?

Do you have Lisp (or other functional programming languages) programming experience?

Do you have experience in proving theorems in the propositional calculus?

Based on what you know so far about CS340d are you are ready to go?

University Experience

Based on your past experience with UTCS faculty and courses, select the number below of the statement that reflects the closest to how you feel about being a part of the UTCS community? (select one answer from below)

1. What is UTCS?
2. I don't feel that UTCS welcomes students.
3. I have no positive or negative feelings.
4. I feel good about being a part of UTCS.
5. I am a member in good standing of the UTCS community.

Course Expectations

What do you hope to learn from this class? (select one answer from below)

1. I am not exactly sure what this course is about.
2. How I can improved my ability to write correct code given user requirements.
3. The theoretical underpinnings of a mechanized logic.
4. How to use ACL2 to formally verify a digital system.
5. Using ACL2 across the fields of mathematics, software, hardware and systems.

What grade do you expect to achieve in CS340d? (select one answer from below)

1. Don't know.
2. C
3. B
4. B+

5. A

What are your concerns, if any, about CS340d? (select one answer from below)

1. No concerns, as I said above I'm ready to go!
2. Some concerns, but I am confident that working together we can have a successful semester in CS340d.
3. Some concerns, but not about the course work itself.
4. Some concerns, mostly about work-load and managing all my classes.
5. I have many concerns about how the class will go for me with the work-load in this challenging course.

If you answered any of 2 - 5 from the previous question, please list some of your top concerns.

-

Other things

Complete the following sentence: ''I wish my professor knew...''

What is one surprising thing about you we would never have guessed? (e.g., Scott worked in the Ford Research Labs on a project to power cars with hydrogen fuel cells and Prof. Hunt visited Scott at the Ford labs and drove a prototype fuel cell car.)

What is the one question we didn't ask on this survey that we should have asked?

Again, please update your information in Canvas and Zoom so you are sure to receive up-to-date course and university information and alerts.

4.2 Quiz 1 Checkout Canvas Quiz Submission

```
Quiz 1
CS 340d
Unique Number: 52285
Spring, 2023

Given:  January 12, 2023
```

Quiz 1: What have you heard so far?

```
Question 1: Indicate for each statement whether it is True or False

     A.)  ACL2 is a system for the automatic memoization of ACL2-Lisp functions.

     B.) (OR (ZP x) (NOT (ZP x)))

     C.) TRACE$  is going to be your debugging friend.

     D.) ACL2 is a formal logic for modeling and reasoning about digital systems.

     E.) ACL2 is an applicative (side-effect-free) subset of Common Lisp
```

Question 2:

Our classroom seems to smell like mold or mildew. Should we ask if another classroom is available?

4.3 Quiz 2 Propositional Calculus

```
                    Quiz 2
                  CS 340d
              Unique Number: 52285
                  Spring, 2023

            Given:  January 26, 2023
```

Quiz 2: Propositional Calculus (PC)

Question 1: The definition of PC operators is often given using Truth Tables TT). For each of the PC operators (logic name, ACL2 function, logic symbol):

- (Conjunction, AND, $\wedge$)
- (Disjunction, OR, $\vee$)
- (Negation, NOT, $\neg$)
- (Implication, IMPLIES, $\Rightarrow$)
- (Equivalence, EQUAL, $\equiv$)

Fill in the Truth Tables below.

A.) (Conjunction, AND, $\wedge$)

P	Q	P /\ Q
T	T	?
F	T	?
T	F	?
F	F	?

B.) (Disjunction, OR, $\vee$)

P	Q	P \/ Q
T	T	?
F	T	?
T	F	?
F	F	?

C.) (Negation, NOT, $\neg$)

P	¬ P
T	?
F	?

D.) (Implication, IMPLIES, $\Rightarrow$)

P	Q	P => Q
T	T	?
F	T	?
T	F	?
F	F	?

E.) (Equivalence, EQUAL, $\equiv$)

P	Q	P == Q
T	T	?
F	T	?
T	F	?
F	F	?

Question 2:

A PC formula is a *theorem*, if its output is true for all values of the input variables. Using the definitions of the PC operators given above, determine, if the following formula is a theorem: $P \wedge Q \Rightarrow P \vee Q$

4.4 Quiz 3 Propositional Calculus

```
Quiz 3
CS 340d
Unique Number: 52285
Spring, 2023

Given:  January 24, 2023
```

Quiz 3: Propositional Calculus (PC): Is this a theorem?

Question 1: The formula p $\wedge$ q $\Rightarrow$ p is a theorem. (true or false)

Question 2: The formula $\neg$(p $\Rightarrow$ q) $\equiv$ ($\neg$q $\Rightarrow$ $\neg$p) is a theorem. (true or false)

Question 3: How could we write the formula p $\wedge$ q $\Rightarrow$ p in ACL2?
(select one)

- ((p and q) implies p)
- (p and q implies p)
- (implies (and (p q)) (p))
- (implies (and p q) p)

Question 4: How could we write the following formula in ACL2 $\neg$($\neg$q $\Rightarrow$ $\neg$p) in ACL2?
(select one)

- (not (not q) implies (not p))
- (not (implies (not q p)))
- (not (implies (not q) (not p)))
- (not not q implies not p)

4.5 Quiz 4 Propositional Calculus

```
Quiz 4
CS 340d
Unique Number: 52285
Spring, 2023

Given:  January 26, 2023
```

Quiz 4: Yet More Propositional Calculus (PC): Is this a theorem?

Question 1: In an axiomatic logic system axioms and inference rules are assumed to be true. They are taken to hold without proof. (true or false)

Question 2: The formula $\neg(p \wedge q) \equiv \neg p \vee \neg q$ is a theorem. (true or false)

Question 3: The formula nil $\Rightarrow$ q is a theorem. (true or false)

Question 4: The formula $p \wedge \neg p$ (called a contradiction) is a theorem. (true or false)

Question 5: The formula $p \vee \neg p$ (called the excluded middle) is a theorem. (true or false)

4.6 Quiz 5 Functional programming in ACL2

```
                    Quiz 5
                  CS 340d
              Unique Number: 52285
                  Spring, 2023

            Given:  February 7, 2023
```

Quiz 5: Basic ACL2

Question 1: We draw the following "ascii-art" for the list (1 2). (true or false)

```
  *
 / \
1   2
```

Question 2: (1 2) is a "cons pair". (true or false)

Question 3: The ACL2 expression (quote (+ 1 2 3)) evaluates to (+ 1 2 3). (true or false)

Question 4: The ACL2 expression (equal (quote (+ 1 2 3)) '(+ 1 2 3)) evaluates to T. (true or false)

Question 5: The ACL2 expression (car '(((a b (c)) d e (f)))) evaluates to. (select one)

- a
- (a b (c))
- ((a b (c)) d e (f))
- NIL
- (((a b (c)) d))

4.7 Quiz 6 Functional programming in ACL2

```
Quiz 6
CS 340d
Unique Number: 52285
Spring, 2023

Given:  February 9, 2023
```

Quiz 6: Questions about ACL2 Terms

Question 1: Which of the following is not a nil-terminated list. (select one)

- (1 2 3)
- (1 2 . nil)
- (1 2 . 3)
- All of the above
- None of the above

Question 2: The COND macro is often used in lisp programming. This code will evaluate to which of the following. (select one)

```
(let ((x 10))
  (cond ((atom x) x)
        ((consp x) (car x))
        (t nil)))
```

- nil
- (car 10)
- 10
- (10)
- None of the above.

Question 3: This code will evaluate to which of the following. (select one)

```
(let ((x '(9 8 7 6)))
  (cond ((atom x) x)
        ((consp x) (car x))
        (t nil)))
```

- nil
- (9 8 7 6)
- 9

- (9)
- None of the above.

Question 4: This code will evaluate to which of the following. (select one)

```
(let ((x "str"))
  (cond ((atom x) x)
        ((consp x) (car x))
        (t nil)))
```

- nil
- x
- "str"
- ("str")
- None of the above.

Question 5: The ACL2 expression '(+ 1 NIL 2 NIL 3 NIL) evaluates to. (select one)

- 1
- (+ 1 NIL 2 NIL 3 NIL)
- 10
- NIL
- This produces a hard-error in ACL2 and enters a break-loop

4.8 Quiz 7 A Quiz Poll

Quiz 7
CS 340d
Unique Number: 52285
Spring, 2023

Given: February 9, 2023

The class was polled on the difficulty of quiz 6. The results were 40% felt it was too difficult, and 60% did not feel it was too difficult. Thank you for your feedback.

4.9 Quiz 7a An ACL2 Lisp Function

Quiz 7a
CS 340d
Unique Number: 52285
Spring, 2023

Given: February 14, 2023

Quiz 7a: An ACL2 Lisp function.

As was indicated in the course materials and emphasized in the class lectures, you will be working in a subset of Common Lisp this semester. Since some of you indicated in the survey that you are new to Lisp, we will help you along with Lisp quizzes from time to time. This is such a time.

Consider the following lisp function, which ACL2 admits into the logic.

```
(defun x (x)
   (if (consp x)
         (cons (len x)
                     (x (cdr x)))
     nil))
```

What do each of the following function calls return?

ACL2 !> (x nil)

ACL2 !> (x 10)

ACL2 !> (x '(a b c d))

ACL 2 !> (x '(10 #\a "string" (22/7)))

ACL2 !> (x '(a . (b . c)))

4.10 Quiz 8 Terms

Quiz 8
CS 340d
Unique Number: 52285
Spring, 2023

Given: February 16, 2023

Quiz 8: Terms.

Let g be a function with signature

```
  (g * * *) => *
```

i.e., g has arity 3 and returns a single value.

Also recall that (equal '3 3) returns T (and similarly for other numbers).

Q1: Which of the following are terms?

1. (g (car x) (cons (g x y z) z) z)
2. (g (car '2) x z)
3. (g '1 x (cons z))
4. (car (g '1 '2 '3))

Q2: Which of the following is the result of the substitution

{x <- (car x), y <- (g '1 '2 '3), z <- (g x y w)} applied to (g x y z):

1. (g (car x) (g '1 '2 '3) (g x y w))
2. (g (car x) (g '1 '2 '3) (g (car x) y w))
3. (g (car x) (g '1 '2 '3) (g x (g '1 '2 '3) w))
4. (g (car x) (g '1 '2 '3) (g (car x) (g '1 '2 '3) w))

4.11 Quiz 9 Dot Notation

```
                    Quiz 9
                  CS 340d
              Unique Number: 52285
                  Spring, 2023

            Given:  February 21, 2023
```

Quiz 9: Dot Notation.

```
Q1:  Which of the following expressions are equivalent to the expression ‘(A B C)?
     Select all that apply.

       1. '(A . (B . (C)))

       2. '(A . (B . (C . nil)))

       3. '(A . (B C))

       4. '(A B (C . nil))

       5. All the Above


Q2. Identify these equivalence statements as TRUE or FALSE.

        __ '(A B . C D) == '(A (B . C) (D . nil))

        __ '(nil (nil nil)) == '(nil (nil))

        __ '(A (B . C) D) == '(A (B . C) . (D . nil))

        __ '(nil . (A . (B . C))) == '(nil A B . C)

        __ '((nil . A) (nil . B) C) == '((nil . A) (nil . B) . (C))
```

4.12 Quiz 10 More on Terms

```
                    Quiz 10
                  CS 340d
               Unique Number: 52285
                   Spring, 2023

             Given:  February 23, 2023
```

Quiz 10: More on Terms.

Terms are an important concept in ACL2.

So far we have covered the definition of Terms; we have had homework on identifying Terms; we have discussed abbreviations, substitutions and the relationship between Terms and formulas.

This quiz should help you gauge how you are coming along in your understanding of terms.

```
Let h be a function with signature

  (h * *) => *

i.e. h has arity 2 and returns a single value.


Also recall that (equal '3 3) returns T (and similarly for other numbers).

Q1:  Which of the following are terms?

    1. (if (car (car x)) (cons 't x) '0)

    2. (h (car '2) x (cdr z))

    3. (car (cons x y) 'nil v)

    4. (car (h '1 '2 ))

    5.  (IF  (IF  X 'T 'NIL) Y  Z)


Q2:  Which of the following is the result of the substitution

{ x <- (first x),  y <- (h '1 '2) }  applied to  (h x y):

    1. (h (car x) (h '1 '2))
```

```
   2. (h (cons (car x) (h '1 '2)))

   3.  nil

   4. (h (first x) (h '1 '2 ) )

   5. (h (car x) (h '1 '2 '3) (h (car x) (h '1 '2 '3) w))
```

```
Q3:  Which of the following terms evaluate to nil

   1. (equal ' "Hello"  'Hello)

   2. (binary-+ 'nil '3)

   3. (car 'nil)

   4. (if 'nil '1 '2)

   5. (equal (cons '3 '4) (cons 'three 'four))
```

4.13 Quiz 11 The Definitional Principle

```
                    Quiz 11
                  CS 340d
               Unique Number: 52285
                   Spring, 2023

              Given:  February 23, 2023
```

Quiz 11: The Definitional Principle.

Indicate whether the following statements are True or False.

```
Q1: The following function meets the Definitional Principle. (True/False)

(defun F (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (F e x))
    nil))


Q2: The following function meets the Definitional Principle. (True/False)

(defun G (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (G e (car x)))
    nil))

Q3: The following function meets the Definitional Principle. (True/False)

(defun H (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (H h (cdr x)))
    nil))

Q4: The following function H has x as a measured formal. (True/False)

(defun H (e x)
  (if (consp x)
```

```
  (if (equal e (car x))
      t
    (H e (cdr x)))
nil))
```

4.14 Quiz 12 Concepts Review

```
                    Quiz 12
                  CS 340d
              Unique Number: 52285
                  Spring, 2023

            Given:  February 23, 2023
```

Quiz 12: Concepts Review.

```
Q1: Which of the following are true-listp?

    1. '(a b c)

    2. '(a b . c)

    3. ()

    4. '((a . b) c)

    5. '( () . nil)

    6. '(a . b c)


Q2:  Which of the following are terms?

    1. (if (car (car x)) (cons 't x) '0)

    2. (h (car '2) x (cdr z))

    3. (car (cons x y) 'nil v)

    4. (car (h '1 '2 ))

    5.  (IF  (IF  X 'T 'NIL) Y  Z)


Q3:  In Homework-4 we proved that: (equal (if 6 x y) x) is a theorem.
     Using Axiom 2, and the substitution sigma = { x <-- 6, y <-- x, z <-- y}
     we get the following theorem:
```

1. $6 \neq nil \Rightarrow$ (if 6 x z) = x
2. $6 \neq nil \Rightarrow$ (if 6 x y) = y

3. $t \neq nil$
4. $6 \neq nil \Rightarrow$ (if 6 x y) = x
5. None of the above

Q4: Q3 above is an example of using: (Select all that apply)

1. axiomatization
2. substitution
3. instantiation
4. referenciation
5. equalization

4.15 Quiz 13 Prove it

```
                          Quiz 13
                        CS 340d
                    Unique Number: 52285
                        Spring, 2023

                   Given:  February 23, 2023


Quiz 13: Prove it!

The following function showed up in Lecture 7 where we
were checking to see if it meets the Definitional Principle.
We all agreed that it did meet the 4 requirements.

(defun f (x)
      (if (consp x)
          (f (cdr x))
         t))

ACL2 has a function called booleanp which is defined as follows.

(defun booleanp (x)
       (if (eq x t) t (eq x nil)))

Prove the following conjecture ψ: (booleanp (f x))

Perhaps we can prove it by induction.

Base Case:

(implies (not (consp x))
         (booleanp (f x)))

< Def, f >

(implies (not (consp x))                    ; hyp1
        (booleanp (if (consp x)
                      (f (cdr x))
                       t)))

< hyp1 and Axiom 3>

(implies (not (consp x))                    ; hyp1
         (booleanp T))
```

```
< Def. booleanp  and Axiom 2>

T
```

QED

This completes the proof of the Base Case. Now for proof of the Induction Step.

```
Induction Step:

Choose induction variable:  x, and the
car/cdr substitution (sigma):  σ1 = (x (cdr x))

Now prove:
(implies (and (consp x)                    ; hyp1
              (booleanp (f (cdr x)))       ; hyp2 (IH)
                 )
               (booleanp (f x)))

< Def. f >

(booleanp (if (consp x)
               (f (cdr x))
               t))

< hyp1 and Axiom 2 >

(booleanp (f (cdr x)))

< But this IS hyp2, the (IH) >

T
```

QED

This completes the proof of the Induction Step. Therefore by the Principle of Structural Induction we have proved the conjecture ψ.

4.16 Quiz 14 Prove it

```
                                Quiz 14
                              CS 340d
                          Unique Number: 52285
                              Spring, 2023

                        Given:  March 23, 2023


From Quiz 14.

In lecture we defined nat as

(defun nat (x)
  (if (consp x)
      (and (equal (car x) nil)
           (nat (cdr x)))
      (equal x nil)))

Now we define min-nat that takes in two arbitrary
lists and return the nat representing the length of the shorter list.

(defun min-nat (x y)
  (if (consp x)
      (if (consp y)
          (cons nil (min-nat (cdr x) (cdr y)))
          nil)
      nil))

Prove the following:

(defthm nat-min-nat
    (nat (min-nat x y)))

Let's attempt the proof by induction.

Let's first prove the base case

(implies (not (consp x))
         (nat (min-nat x y)))

By the definition of min-nat and substitution, we have

(implies (not (consp x))  ; hyp 1
         (nat (if (consp x)
                  (if (consp y)
```

```
                    (cons nil (min-nat (cdr x) (cdr y)))
                    nil)
                nil)))
```

By hyp 1 and axiom 3, we have

```
(implies (not (consp x))  ; hyp 1
         (nat nil))
```

By definition of nat, axiom 8, and axiom 3, we have

```
(implies (not (consp x))  ; hyp 1
         (equal nil nil))
```

By axiom 5, we have T.

We've proven the base case:

```
(implies (not (consp x))
         (nat (min-nat x y))).
```

Now let's move on to the inductive step.

Choose induction variable x, and substitution \phi = {(x (cdr x)), (y (cdr y))}.

For the inductive step, we need to prove:

```
(implies (and (consp x)                          ; hyp 2
              (nat (min-nat (cdr x) (cdr y)))) ; IH (inductive hypothesis)
         (nat (min-nat x y))).
```

By definition of min-nat, we have

```
(implies (and (consp x)                            ; hyp 2
              (nat (min-nat (cdr x) (cdr y))))   ; IH (inductive hypothesis)
         (nat (if (consp x)
                  (if (consp y)
                      (cons nil (min-nat (cdr x) (cdr y)))
                      nil)
                  nil)))
```

By hyp 2 and axiom 2, we have

```
(implies (and (consp x)                                  ; hyp 2
              (nat (min-nat (cdr x) (cdr y))))        ; IH
```

```
         (nat (if (consp y)
                  (cons nil (min-nat (cdr x) (cdr y)))
                  nil))).
```

Now let's do a case split on y: (not (consp y)) and (consp y).

Consider the case where (not (consp y)):

```
(implies (and (consp x)                          ; hyp 2
              (not (consp y))                    ; hyp 2a
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (nat (if (consp y)
                  (cons nil (min-nat (cdr x) (cdr y)))
                  nil)))
```

By hyp 2a and axiom 3, we have

```
(implies (and (consp x)                          ; hyp 2
              (not (consp y))                    ; hyp 2a
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (nat nil))
```

Following what we did in the base case, we will reach T eventually.

Now onto the case where (consp y):

```
(implies (and (consp x)                          ; hyp 2
              (consp y)                          ; hyp 2b
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (nat (if (consp y)
                  (cons nil (min-nat (cdr x) (cdr y)))
                  nil)))
```

By hyp 2b and axiom 2, we have

```
(implies (and (consp x)                          ; hyp 2
              (consp y)                          ; hyp 2b
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (nat (cons nil
                    (min-nat (cdr x) (cdr y))))).
```

```
(defun nat (x)
  (if (consp x)
      (and (equal (car x) nil)
           (nat (cdr x)))
      (equal x nil)))


By definition of nat, we have
(implies (and (consp x)                          ; hyp 2
              (consp y)                          ; hyp 2b
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (if (consp (cons nil (min-nat (cdr x) (cdr y))))
             (and (equal (car (cons nil (min-nat (cdr x) (cdr y)))) nil)
                  (nat (cdr (cons nil (min-nat (cdr x) (cdr y))))))
             (equal (cons nil (min-nat (cdr x) (cdr y)))
                    nil)))


By axiom 9, and axiom 10, we have
(implies (and (consp x)                          ; hyp 2
              (consp y)                          ; hyp 2b
              (nat (min-nat (cdr x) (cdr y))))   ; IH
         (if (consp (cons nil (min-nat (cdr x) (cdr y))))
             (and (equal nil nil)
                  (nat (min-nat (cdr x) (cdr y))))
             (equal (cons nil (min-nat (cdr x) (cdr y)))
                    nil)))


By axiom 2 and axiom 7, we have
(implies (and (consp x)                            ; hyp 2
              (consp y)                            ; hyp 2b
              (nat (min-nat (cdr x) (cdr y))))     ; IH
         (and (equal nil nil)
              (nat (min-nat (cdr x) (cdr y)))))



By IH, axiom 5, we have
(implies (and (consp x)                          ; hyp 2
              (consp y)                          ; hyp 2b
              (nat (min-nat (cdr x) (cdr y)))) ; IH
         (and T T)),

and we'll reach T and hence prove our theorem.
```

4.17 Quiz 15 Prove it

Quiz 15
CS 340d
Unique Number: 52285
Spring, 2023

Given: April 23, 2023

This page is intentionally blank. Quiz 15 does not exist, due to a numbering error. Sorry about that.

4.18 Quiz 16 Is this a defthm

```
                                Quiz 16
                              CS 340d
                          Unique Number: 52285
                              Spring, 2023

                        Given:  April ??, 2023


From Quiz 16: Is this a defthm?

Consider the definitions below for the following questions.
You can use knowledge from previous lectures and/or ACL2 to help you answer the questions.

(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
    y))


(defun rev (x)
  (if (atom x)
      nil
    (app (rev (cdr x))
         (list (car x)))))


(defun true-listp (x)
  (if (consp x)
      (true-listp (cdr x))
    (eq x nil)))


1. Is this a theorem? (true or false)

(defthm true-listp-rev
  (true-listp (rev x)))

2. Is this a theorem? (true or false)

(defthm rev-of-rev
  (implies (true-listp x)
           (equal (rev (rev x))
                  x)))
```

```
3. Do we need the true-listp hypothesis in the
    previous theorem? I.e. is this a theorem?

(defthm rev-of-rev-2
  (equal (rev (rev x))
         x))


4. Is this a theorem? (true or false)

(defthm rev-of-rev-of-rev
  (equal (rev (rev (rev x)))
         (rev x)))
```

4.19 Quiz 17 The Method

```
                          Quiz 17
                        CS 340d
                    Unique Number: 52285
                        Spring, 2023

                   Given:  April 13, 2023

From Quiz 17: The Method

Consider the following definitions for representing natural numbers
as lists of nils and a "plus" operation to add these numbers.

(defun nat (x)
  (if (atom x)
      (null x)
    (and (null (car x))
         (nat (cdr x)))))

(defun make-nil-list (x)
  (if (atom x)
      nil
    (cons nil (make-nil-list (cdr x)))))

(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

(defun plus (x y)
  (app (make-nil-list x)
       (make-nil-list y)))



These will be used in the following questions. You may use
lecture knowledge and ACL2 to answer the questions.


1. Is this a theorem? (true or false)

(defthm nat-plus
  (nat (plus x y)))

2. Provide the following defthm event into ACL2. Does it prove?
```

```
    (true or false)

(defthm associativity-of-plus
  (equal (plus (plus i j) k)
         (plus i (plus j k))))
```

3. What is the key checkpoint printed by ACL2?
Copy and paste the checkpoint into the box below.

4. In this case, we may need some key insight to help ACL2 along.
One such fact is the theorem below. Provide it to ACL2.

```
(defthm make-nil-list-fact
  (equal (make-nil-list (app x y))
         (app (make-nil-list x)
              (make-nil-list y))))
```

Now, provide the associativity-of-plus theorem to ACL2 again.
Does the theorem prove?
(true or false)

5. Note the key checkpoint from the last proof attempt of associativity-of-plus.
If needed, what theorem do you think will help?
Now, enter the following theorem into ACL2.
Does this help prove the associativity-of-plus theorem? (true or false)

```
(defthm make-nil-list-twice
  (equal (make-nil-list (make-nil-list x))
         (make-nil-list x)))
```

5 CS340d Homework

Homework problems are designed to solidify your understanding of various concepts related to this course. Problems may appear here prior to their assignment. We reserve the right to alter homework assignment up to the date of assignment. Why? We may not be able to cover in class everything we hoped to discuss prior to some specific date; this, in turn, will affect when we expect students to be able to respond to CS340d homework assignments.

5.1 Homework 0

```
Homework Assignment 0
CS 340d
Unique Number: 52285
Spring, 2023

Given: January 12, 2023
Due:  January 19, 2023
```

Part 1:

Part 1 of your homework assignment is to familiarize yourself with the documentation system we will be using for ACL2. This is a web-based system that you can access from the following link.

`acl2.org/manual`

We recommend that students begin by starting a browser and loading: http://acl2.org/manual – a Webpage should be displayed. Then, in the "Jump-to" dialogue box, we recommend that a student enter "gentle-intro-to-acl2-programming". Note, most browsers will attempt to auto complete. A student should read and try to understand everything in that webpage. Please, read through the section titled "Symbols."

ACL2 lisp operates by what is called a read-eval-print loop. At the ACL2 prompt you enter the form you want evaluated and ACL2 reads that input, evaluates it and prints out the result of the evaluation. The ACL2 prompt contains a lot of information about the state of ACL2. What do the following ACL2 prompts tell you about the state of the system?

```
1.)  ACL2 !>

2.)  ACL2 >

3.)  ACL2 p!>

4.)  ACL2 p>
```

Use the reading you did in "A Gentle Introduction to ACL2 Programming," and the ACL2 documentation system to answer. For extra credit, what do these prompts mean?

```
5.)  KEYWORD >

6.)  ACL2 !s>
```

Part 2:

We will be using the ACL2 Lisp programming language immediately in this course, and the theorem-proving system starting in a couple of weeks. To get started you need to confirm that you can access and run the ACL2 system on the UTCS Linux cluster, and on your personnel machine. Remember, we will be working in every class using ACL2.

The top-level ACL2 webpage can be found here:

`http://acl2.org/manual`

If you wish to use this system on UTCS Linux-based computers, then be sure that you have appropriate user usage privileges. Once you have logged into a UTCS Linux-based computer, then you may type:

/p/bin/acl2

at the unix system prompt to start ACL2 running. If you wish to use ACL2 on your personal laptop, then you will need to follow the ACL2 installation instructions which can be found here:

`http://www.cs.utexas.edu/users/moore/acl2/v7-0/HTML/installation/installation.html`

Before ACL2 can be built on your laptop or deskside computer, you will need to obtain a Lisp implementation. We strongly recommend that you use Clozure Common Lisp. Why? This is the Lisp that we generally use. This hashtag contains a bit of information that might be helpful.

`http://www.cs.utexas.edu/users/moore/acl2/v7-0/HTML/installation/requirements.html#Obtaining-CCL`

Part 2 of your homework assignment is to provide us with a copy of the ACL2 welcome header you see when you run the command to start-up ACL2 on the computer you will be using for ACL2 assignments this semester.

It should look something like this...

```
sms@Scott-MacBook-Pro src % ~/ccl-acl2-8.4/saved_acl2
```

Welcome to Clozure Common Lisp Version 1.12 (v1.12-32-g8778079b) DarwinX8664!

```
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 + ACL2 Version 8.4                                                         +
 +   built March 21, 2022  08:08:16.                                        +
 + Copyright (C) 2021, Regents of the University of Texas.                  +
 + ACL2 comes with ABSOLUTELY NO WARRANTY.  This is free software and       +
 + you are welcome to redistribute it under certain conditions.  For        +
 + details, see the LICENSE file distributed with ACL2.                     +
 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

System books directory "/Users/sms/acl2/ccl-acl2-8.4/books/".
Type :help for help.
```

```
Type (quit) to quit completely out of ACL2.

;;;
;;; Quick test of defining a recursive function in ACL2
;;;
ACL2 !> (defun sum-to-n (n)
              (if (zp n) ; Is n zero?
                 0
               (+ n
                  (sum-to-n (1- n)))))

The admission of SUM-TO-N is trivial, using the relation O< (which
is known to be well-founded on the domain recognized by O-P) and the
measure (ACL2-COUNT N).  We observe that the type of SUM-TO-N is described
by the theorem (AND (INTEGERP (SUM-TO-N N)) (<= 0 (SUM-TO-N N))).
We used the :compound-recognizer rule ZP-COMPOUND-RECOGNIZER and primitive
type reasoning.

Summary
Form:  ( DEFUN SUM-TO-N ...)
Rules: ((:COMPOUND-RECOGNIZER ZP-COMPOUND-RECOGNIZER)
        (:FAKE-RUNE-FOR-TYPE-SET NIL))
Time:  0.00 seconds (prove: 0.00, print: 0.00, other: 0.00)
 SUM-TO-N
ACL2 !> (good-bye)

sms@Scott-MacBook-Pro ccl-acl2-8.4 %
```

NOTE: If you have any trouble getting access to ACL2 on the UTCS Linux machines, or difficulty in installing and running ACL2 on your personal machine, see Professor Hunt or one of his staff immediately. You cannot do the work for this class without this software system available to you.

5.2 Homework 1

```
Homework Assignment 1
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  January 19, 2023
 Due:   January 26, 2023
```

This homework assignment considers further our choice of ACL2 as the programming language for cs340d. If you search around for information on the most commonly used programming languages in industry, you will often find a list like the following for the top five or six.

```
Python
JavaScript
Java
C#
C
C++
```

You have Python, which according to Peter Norvig is a dialect of Lisp. See more detail on that claim here (`https://norvig.com/python-lisp.html`).

Many of the languages that are variants of C. C# and C++ added object-oriented programming into C. Java might be consider as a newer version of C designed with network (read internet) programming in mind.

To do the types of verification that we have in mind we need a formally-defined programming language and a verifier for that logic. We will use ACL2. Note, there are other systems like ACL2; for example, NuPRL, Coq, HOL, PVS. We will not investigate these systems, but they are all interesting.

We will explore further the SUM-TO-N function, and compare it to an output equivalent function in "your" favorite programming language. The specification, in natural language, is:

```
A program that when given a Natural number, n, computes and returns
the sum of the Natural numbers from 1 up and including n.
```

$\sum_{k=1}^{n} k$

```
We offer this function.  Does it meet its specification?

(defun sum-to-n (n)
  (declare (xargs :guard (natp n)))
  (if (zp n) ; Is n zero?
      0      ; yes, return 0
      (+ n   ; no,  recursive call to sum-to-n
         (sum-to-n (1- n)))))
```

Part 1:

Continue reading in "gentle-introduction-to-ACL2-programming" through the section titled Common Patterns of Recursion.

Part 2:

2.1 Write a python program that is output equivalent to the ACL2 program for SUM-TO-N.

2.2 Write a C or Java program that is output equivalent to the ACL2 program for SUM-TO-N.

Part 3:

3.1 Defend the correctness of the programs you have written. How can you verify or prove that your code satisfies its specification? For the ACL2 version, we can use the ACL2 theorem prover to formally verify its correctness.

```
(defthm sum-to-n-returns-natp
  (implies (natp n)
           (natp (sum-to-n n))))
```

In fact, it doesn't matter if the input is a natural number.

```
(defthm sum-to-n-returns-natp-2
  (natp (sum-to-n n)))
```

We know from Gauss formula the the sum of the Natural numbers up to n is given by $\frac{n(n+1)}{2}$. So we will take that as the specification of sum-to-n.

```
(defun sum-to-n-spec (n)
  (declare (xargs :guard (natp n)))
  (/ (* (1+ n) n) 2))

; Now state the theorem (Main Result) that must be true
; if sum-to-n meets its specification.

(defthm correctness-of-sum-to-n
  (implies (natp n)
           (equal (sum-to-n n)
                  (sum-to-n-spec n))))
```

Part 4:

Using ACL2, write solutions to these challenges:

1. Define a function, MULTIPLY, to multiply all of the numbers in a list.

2. Define a function, SUM-TIPS, to sum all tips of a tree with integers at the tips.

3. Define the FLATTEN function, which recursively appends the FLATTEN of a left subtree to the FLATTEN of a right subtree.

4. Define a flatten function which doesn't use APPend, but just uses CONS.

5. Argue that FLATTEN and the function from 4. produce the same result.

5.3 Homework 2

```
Homework Assignment 2
CS 340d
Unique Number: 52285
Spring, 2023

Given:  January 26, 2023
Due:    February 9, 2023
```

This homework concerns the 5 data types in ACL2: numbers, characters, strings, symbols, and ordered pairs.

Part 1:

Continue reading in "gentle-introduction-to-ACL2-programming" through the section titled Functions on Binary Trees.

```
Problem 1.
Each of the utterances below is supposed to be a single object.
Say whether it is a number, string, symbol, pair, or ill-formed
i.e., does not represent a single object in our language).

1.   Monday
2.  π
3.  HelloWorld!
4.  --1
5.  -1
6.  *PI*
7.   31415x10**-4
8.   (A . B . C)
9.   Hello World!
10.  if
11.  invokevirtual
12.  ((1) . (2))
13.  <=
14.  ((A . 1) (B . 2) (C . 3))
15.  Hello_World!
16.  +
17.  lo-part
18.  3141592653589793238462643383279502884197169399375105820974944592 3
19.  (1 . (2 . 3))
20.  (1 . 2 3)
21.  "Hello World!"
22.  ((1) (2) . 3)
23.  ()
```

```
Problem 2.
Group the constants below into equivalence classes. That is,
some items below are equal to others even though they are
displayed differently; group equal constants together.

1.   (1 . (2 3))
2.   (nil . (nil nil))
3.   ((nil nil) . nil)
4.   (1 (2 . 3) 4)
5.   (nil nil)
6.   (1 (2 . 3) . (4 . ()))
7.   (HelloWorld !)
8.   (1 (2 3 . ()) 4)
9.   ((A . t) (B . nil)(C . nil))
10.  (()())
11.  (1 2 3)
12.  (() () . nil)
13.  (A B C)
14.  (a . (b . (c)))
15.  (HELLO WORLD !)
16.  ((a . t) (b) . ((c)))
```

ACL2 provides built-in type functions that recognize the ACL2 data types. They are named acl2-numberp, characterp, stringp, symbolp and consp respectively.

Note that each of these function names end in the letter p. This is an informal Lisp naming convention for symbols naming predicates.

A predicate is a boolean function. That is a function that returns either T or NIL. You can define a function booleanp which recognizes the two boolean values as follows.

```
(defun booleanp (x)
       (declare (xargs :guard t))
       (if (eq x t) t (eq x nil)))
```

You can say that s is of type boolean, if (booleanp s) returns T. You can also state a theorem (thm) that describes an important property of the function booleanp as follows.

```
(thm (or (booleanp x) (not (booleanp x))))
```

Enter this theorem into ACL2. Is it a theorem? What is the name of the above theorem?

Therefore you can define any type of object you need to support a system model by defining a recognizer (type) for that object. For example, if you are working on Natural numbers, you might have a recognizer called (natp x) which you could define as follows.

```
(defun natp (n)
       (and (integerp n) (<= 0 n)))
```

Note that some math types are also built-in functions in ACL2. For example, all of the following are built-in for math: integerp, rationalp, complex-rationalp.

Part 2:

```
Define functions for the following predicates in ACL2.

(plusp x) - returns T if x is a positive number, otherwise NIL.

(minusp x) - returns T if x is a negative number, otherwise NIL.

(natp x) - returns T if x is a natural number, otherwise NIL.

(posp x) - returns T if x is a positive integer, otherwise NIL.

(evenp x) - returns T if x is an even integer, otherwise NIL.

(oddp x) - returns T if x is an odd integer, otherwise NIL.

(zerop x) - returns T if x is zero, otherwise NIL.
```

Check your definitions by running them in ACL2.

```
To get a high-level overview of the partition of numbers
supported in ACL2 study the following graphic and
classifying function.

ACL2 Numbers
 |
 |-- Rationals
 |    |
 |    |-- Integers
 |    |     |
 |    |     |-- Positive integers         3
 |    |     |-- Zero                      0
 |    |     |-- Negative Integers        -3
 |    |
 |    |-- Non-Integral Rationals
 |    |     |
 |    |     |-- Positive Non-Integral Rationals   19/3
 |    |     |-- Negative Non-Integral Rationals   -22/7
 |
 |-- Complex Rational Numbers           #c(3 5/2) ; i.e., 3 + (5/2)i

(defun classify-number (x)
  (cond ((rationalp x)
         (cond ((integerp x)
                (cond ((< 0 x) 'positive-integer)
                      ((= 0 x) 'zero)
                      (t 'negative-integer)))
               ((< 0 x) 'positive-non-integral-rational)
```

```
       (t 'negative-non-integral-rational)))
((complex-rationalp x) 'complex-rational)
(t 'NaN)))
```

5.4 Homework 3

```
Homework Assignment 3
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  February 7, 2023
  Due:  February 16, 2023
```

This homework will help introduce the ACL2 Logic. It concerns the ACL2 notions of terms and function definitions.

Part 1:

Continue reading in "gentle-introduction-to-ACL2-programming" through the section titled Conclusion. Continue reading in "recursion-and-induction" through the section titled Function Definitions. Do this reading before attempting the problems.

```
In ‘‘recursion-and-induction’’ documentation provide solutions for
problems 3-7.
```

Part 2:

```
In ‘‘recursion-and-induction’’ documentation provide solutions for
problems 10 - 15, 17.
```

5.5 Homework 4

```
Homework Assignment 4
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  February 16, 2023
  Due:  February 23, 2023
```

Simple ACL2 Proofs

So far in this course we have focused our discussions mainly on verification of properties of software. In this homework we introduce use of the ACL2 Logic for modeling digital hardware and asking if a hardware design has certain properties. We look at a simple boolean circuit and ask if it implements the specification.

Part 1:

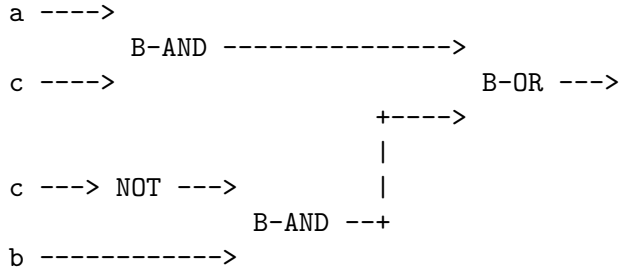
```
Using the same definitions for NOT, B-OR, and B-AND from Part 3 below,
can you demonstrate De Morgan's Law is a validity?
That is, prove the following is a theorem.

  (B-AND x y)  ==  (NOT (B-OR (NOT x) (NOT y)))
```

Part 2:

```
Consider the following expressions.  For each, show that
it is a theorem , or provide a counter example. (and show
your work in detail)

 (equal (if 6 x y) x)


 (equal (if NIL x y) y)


 (equal (if (if x y z) u v)
        (if x (if y u v) (if z u v)))


 (equal (implies x (implies y z))
        (implies (and x y) z))
```

Part 3:

Given the following proposed Boolean circuit:



where:

```
(NOT x)     == (if x NIL T)
(B-OR  x y) == (if x T (if y T NIL))
(B-AND x y) == (if x (if y T NIL) NIL)
```

Consider the specification:

```
  (IF c a b)
```

If the inputs a, b, and c, are constrained to be boolean values, does the circuit diagram (above) implement the specification (IF c a b)?

5.6 Homework 5

```
Homework Assignment 5
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  February 23, 2023
  Due:  March 2, 2023
```

Here is a web pointer to a talk that can help you with this homework. *Hand proof examples by J Moore* (`https://youtu.be/pVRfeu8MbgE`)

This homework will help introduce defining functions in our simplified version of the ACL2 Logic. It concerns the ACL2 notions of terms and function definitions.

Part 1:

You should have completed the reading of "gentle-introduction-to-ACL2-programming" through the section titled Conclusion. Now you should be familiar with some of the common errors that come up during debugging your ACL2/Lisp programs (e.g., Stack overflow, Type errors). You will have read over the common patterns of recursion in function definitions (defun) and become familiar with using the built-in data structure (*cons pairs*) to model lists, trees, a-lists and so on. This material is now fair game for quiz questions, so if you have any lingering questions on the concepts of this part of the course please bring those to class for discussion and clarification.

Continue reading in "recursion-and-induction" through the section titled Structural Induction. Do this reading before attempting the problems.

```
In ``recursion-and-induction'' documentation provide solutions for
problems 27, 28, 29.

Think about problems 27, 28, 29, and write up your thoughts about
these three problems and what impact allowing these definitions might
have on the ACL2 logic.
```

Part 2:

```
In ``recursion-and-induction'' documentation provide solutions for
problems 31-37.
```

NOTE: We (Warren, Vivek, Maxine, Scott) will be available each week at our published office hours. If you don't see us then, come down the hall to our office and ask us for help. In addition, we can be available at other times by appointment. Don't wait until you have a big problem – come see us, and we will try to help you while the problem is still small.

5.7 Homework 6

```
Homework Assignment 6
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  March 2, 2023
  Due:  March 9, 2023
```

Part 1: Tracing

In this part of the homework assignment we look at how tracing can be used to gain familiarity with the data and control flow of two functions that presumably do the same thing. Sometimes you can have two equivalent functions where one is much more efficient (in time or space utilization) while the other is easier to reason (prove theorems) about. If we can show the two functions are equivalent then they can be used interchangeably.

Consider the reverse function which you have seen in an earlier homework. So we are all on the same page we provide this function here, but you are welcome to use an alternative definition that you have written yourself (provided you can show it to be equivalent to the definition below).

```
;; The append function.
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

;; The reverse function - rev
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
      nil))

;; The tail-recursive reverse function - rev1
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
      a))

;; Prove - This is what you will prove in a later homework.

(equal (rev1 x nil) (rev x))
```

Use the trace$ function in ACL2 to understand the operation of both rev and rev1 functions. Write up a short report describing what you have done and what you have learned in this exercise. Include answers to the follow questions in your report.

- Q1: What is the CONS complexity (number of CONS operations) of APP?
- Q2: What is the CONS complexity of REV?
- Q3: What happens if you trace both APP and REV simultaneously?
- Q4: Do you believe that you can prove (equal (rev x) (rev1 x nil)) ?

This will be graded as a writing assignment so be sure and spell check and grammar check your work.

Part 2: Structural Induction

This part of the homework will build your skills in using the key tool for proof in ACL2 – Structural Induction.

Continue reading in "recursion-and-induction" through the section titled Structural Induction. Do this reading before attempting the problems.

Your homework assignment is to provide solutions for problems 40 - 44.

These problems may ask you to prove something that is not a theorem. In this case, provide a counterexample demonstrating that the conjecture is false for some input. Then propose and prove the "intended" theorem. For example, adding a hypothesis can restrict the domain of input so that the conjecture holds.

Furthermore, for some theorems, you may need to state and prove lemmas to aid in the proof process.

Here are some definitions you may wish to use for this (and future) homework assignment(s).

```
(defun tree-copy (x)
  (if (consp x)
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))
    x))

(defun app (x y)
  (if (consp x)
      (cons (car x)
            (app (cdr x) y))
    y))

(defun rev (x)
  (if (consp x)
      (app (rev (cdr x))
```

```
            (cons (car x) nil))
    nil))

(defun mapnil (x)
  (if (consp x)
      (cons nil (mapnil (cdr x)))
    nil))

(defun swap-tree (x)
  (if (consp x)
      (cons (swap-tree (cdr x))
            (swap-tree (car x)))
    x))

(defun mem (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (mem e (cdr x)))
    nil))

(defun sub (x y)
  (if (consp x)
      (if (mem (car x) y)
          (sub (cdr x) y)
        nil)
    t))

(defun int (x y)
  (if (consp x)
      (if (mem (car x) y)
          (cons (car x)
                (int (cdr x) y))
        (int (cdr x) y))
    nil))
```

5.8 Homework 7

```
Homework Assignment 7
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  March 9, 2023
 Due:  March 23, 2023
```

Part 1: Tracing (continued from homework 6)

In homework 6 we looked at how tracing can be used to gain familiarity with the data and control flow of two functions that presumably do the same thing. Sometimes you can have two equivalent functions where one is much more efficient (in time or space utilization) while the other is easier to reason (prove theorems) about. If we can show the two functions are equivalent then they can be used interchangeably.

Consider the reverse function which you have seen in an earlier homework. So we are all on the same page we provide this function here, but you are welcome to use an alternative definition that you have written yourself (provided you can show it to be equivalent to the definition below).

```
;; The append function.
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

;; The reverse function - rev
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
      nil))

;; The tail-recursive reverse function - rev1
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
      a))

;;  Now, based on your work in homework 6,
;;  Prove (by hand proof) - The following if it is is a Theorem.
;;  If it is not a theorem provide a counterexample, and see if
;;  you can tweak it so it becomes a theorem under additional
```

```
;;  assumptions (constraints).
;;

(defthm rev-rev1-equivalence
    (equal (rev1 x nil) (rev x)))

For extra understanding, but no extra credit, see if you
can get ACL2 to prove it.
```

Part 2: Structural Induction

This part of the homework will continue to build your skills in using the key tool for proof in ACL2 – Structural Induction.

Continue reading in "recursion-and-induction" through the section titled Structural Induction. Do this reading before attempting the problems.

Your homework assignment is to provide solutions for problems 45 – 50.

These problems may ask you to prove something that is not a theorem. In this case, provide a counterexample demonstrating that the conjecture is false for some input. Then propose and prove the "intended" theorem. For example, adding a hypothesis can restrict the domain of input so that the conjecture holds.

Furthermore, for some theorems, you may need to state and prove lemmas to aid in the proof process.

NOTE: After this assignment, we will start using the proof builder. This will allow you to accomplish your homework with much less writing.

Here are some definitions you may wish to use for this (and future) homework assignment(s).

```
(defun tree-copy (x)
  (if (consp x)
      (cons (tree-copy (car x))
            (tree-copy (cdr x)))
    x))

(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
    nil))

(defun mapnil (x)
  (if (consp x)
```

```
      (cons nil (mapnil (cdr x)))
    nil))

(defun swap-tree (x)
  (if (consp x)
      (cons (swap-tree (cdr x))
            (swap-tree (car x)))
    x))

(defun mem (e x)
  (if (consp x)
      (if (equal e (car x))
          t
        (mem e (cdr x)))
    nil))

(defun sub (x y)
  (if (consp x)
      (if (mem (car x) y)
          (sub (cdr x) y)
        nil)
    t))

(defun int (x y)
  (if (consp x)
      (if (mem (car x) y)
          (cons (car x) (int (cdr x) y))
        (int (cdr x) y))
    nil))
```

5.9 Homework 8

```
Homework Assignment 8
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  March 23, 2023
  Due:  March 30, 2023
```

Solidify Your Knowledge of the Principle of Structural Induction

This homework is designed to solidify your knowledge of our Principle of Structural Induction by problems that consider reflectivity, commutativity, and transitivity.

Your homework assignment is to provide solutions for problems 51 – 55.

5.10 Homework 9

```
Homework Assignment 9
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  March 30, 2023
  Due:  April 6, 2023
```

Using the (ACL2) Method

This homework is to encourage your use of the ACL2 "Method" for proving theorems about ACL2 conjectures using the ACL2 theorem prover. For an introduction to "The Method", please see the ACL2 Documentation topic: THE-METHOD. This documentation topic will take you to the ACL2 Documentation topic: INTRODUCTION-TO-THE-THEOREM-PROVER. Read through the subtopics:

- INTRODUCTION-TO-REWRITE-RULES-PART-1
- SPECIAL-CASES-FOR-REWRITE-RULES
- EQUIVALENT-FORMULAS-DIFFERENT-REWRITE-RULES
- INTRODUCTION-TO-KEY-CHECKPOINTS
- DEALING-WITH-KEY-COMBINATIONS-OF-FUNCTION-SYMBOLS
- GENERALIZING-KEY-CHECKPOINTS
- POST-INDUCTION-KEY-CHECKPOINTS

which are referenced in this Doc topic.

Part 1: Using the ACL2 Method

Using the ACL2 "The Method,", re-prove the theorems from Homework #6 (Problems 40-44). You may turn in for grading a file of events that lead to proving the theorem for each problem (40 - 44). If you wish, you may use the ACL2 proof builder to create events for the ACL2 system to process, but you must turn an ACL2 script file that can be run by:

cat <your-solution-file.lisp> | acl2

Part 2: The equivalence of rev1 and rev

Prove, with help from the proof-builder or using "The method," the following theorem. Append the ACL2 proof commands you use for this proof with what you created for your Part 1 (above) solution.

```
(equal (rev1 x nil) (rev x))
```

```
The following functions may be helpful, or not.

; The append function.
(defun app (x y)
  (if (consp x)
      (cons (car x) (app (cdr x) y))
    y))

;; The reverse function - rev
(defun rev (x)
  (if (consp x)
      (app (rev (cdr x)) (cons (car x) nil))
      nil))

  (defun mapnil (x)
  (if (consp x)
      (cons nil (mapnil (cdr x)))
    nil))

;; The tail-recursive reverse function - rev1
(defun rev1 (x a)
  (if (consp x)
      (rev1 (cdr x) (cons (car x) a))
      a))
```

5.11 Homework 10

```
Homework Assignment 10
CS 340d
Unique Number: 52285
Spring, 2023

Given:  April 6, 2023
Due:  April 18, 2023
```

More Work with the ACL2 Method

This homework is intended to introduce you to Peano arithmetic.

Your homework assignment is to provide solutions for problems 67 – 73 using ACL2 and "The Method." You will turn in the events file that confirms you proof works. This file must run without error when you turn it in for grading.

If you wish, you may use the ACL2 proof builder to create events for the ACL2 system to process, but you must turn an ACL2 script file that can be run by:

cat <your-solution-file.lisp> | acl2

5.12 Homework 11

```
Homework Assignment 11
       CS 340d
 Unique Number: 52285
     Spring, 2023

Given:  Not Assigned
  Due:  Not Assigned
```

Under construction. This assignment requires ACL2 more general definitional principle. This is not part of the CS340d course requirements.

This homework concerns developing facility in the documentation system available with ACL2, and exploring the ACL2 system interface and interaction style (read-eval-print loop). We will explore further the SUM-TO-N function, and build upon its basic structure as a template or proof strategy that is reusable. We have seen the SUM-TO-N function in homework 0 and in the lectures.

General Comment

Before we develop this homework assignment further, we describe our philosophy for all our assignments and for many of our assignments (quizzes, labs, special projects) in this course. We expect our programs to implement their requirements with mathematical precision, but programs are generally specified with natural language. To this point in your education, most programming assignments have included some description of what program you should write, and then, you are expected to interpret the documentation and produce a result. It requires tremendous care and precision to write a precise description of any computation in a natural language – it is certainly beyond our ability to write completely precise, natural-language specifications.

We would like to write mathematical specifications, and that will require us learn certain mathematics throughout the course of this semester. As a community of software developers, this approach would be extremely valuable where it can be deployed, but it is not yet a mature discipline. Even so, we will sometimes refer to programs that can be specified formally. Like, for example, SUM-TO-N.

Homework 1

- Prove $\sum_{k=1}^{n-1} k(n-k)^2 = \frac{n^2(n^2-1)}{12}$

- Given

  From lectures and class notes we have seen a proof approach with functions INC, SUM-TO-N and SUM-SQ-TO-N. Using this same approach write a function SUM-CUBE-TO-N that sums up the cubes of the numbers 1 to n.

- Prove

  Using ACL2 and the function SUM-SQ-TO-N as a proof strategy, show that this summation is equivalent to:

  $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$

- Lemmas

  With the function SUM-CUBE-TO-N proven, we now have the following three lemmas available to prove the Main Result.
- Lemma-1

  $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

- Lemma-2

  $\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$

- Lemma-3

  $\sum_{k=1}^{n} k^3 = \frac{n^2(n+1)^2}{4}$

Use this information to prove the Main Result by hand, and using ACL2. To be clear, you will submit the following items for grading.

- Your function SUM-CUBE-TO-N.
- Your ACL2 Proof of Lemma-3.
- Your hand-proof of the Main Result.
- Your ACL2 proof of the Main Result.

SOLUTION

The following relationships about summations will be needed.

- Lemma-4

  $\sum_{k=1}^{n-1} k = (\sum_{k=1}^{n} k) - n$

- Lemma-5

  $\sum_{k=1}^{n-1} k^2 = (\sum_{k=1}^{n} k^2) - n^2$

- Lemma-6

  $\sum_{k=1}^{n-1} k^3 = (\sum_{k=1}^{n} k^3) - n^3$

Proof of the Main Result.

This proof is written in a calculational style. It is a series of math steps that rewrite the LHS of the equation into the RHS. Between each step is a justification supporting the validity of the step. The justification in written in <...> brackets. We start with the LHS

$\sum_{k=1}^{n-1} k(n-k)^2$

= < Expand the squared term. >

$\sum_{k=1}^{n-1} k(n^2 - 2kn + k^2)$

$=$ < Distribute k. >

$\sum_{k=1}^{n-1}(kn^2 - 2k^2n + k^3)$

$=$ < Distribute summation operation. >

$n^2 \sum_{k=1}^{n-1} k - 2n \sum_{k=1}^{n-1} k^2 + \sum_{k=1}^{n-1} k^3$

$=$ < Substitute using Lemma-1 through Lemma-6. >

$n^2(\frac{n(n+1)}{2} - n) - 2n(\frac{n(n+1)(2n+1)}{6} - n^2) + (\frac{n^2(n+1)^2}{4} - n^3)$

$=$ < Expand products. >

$\frac{n(n^2)(n+1)}{2} - n^3 - \frac{2(n^2)(n+1)(2n+1)}{6} + 2n(n^2) + \frac{n^2(n+1)^2}{4} - n^3$

$=$ < Factor out n^2 and simplify. >

$n^2(\frac{n(n+1)}{2} - \frac{2(n+1)(2n+1)}{6} + \frac{(n+1)^2}{4})$

$=$ < Put on common denominator. >

$n^2(\frac{6n(n+1)}{12} - \frac{4(n+1)(2n+1)}{12} + \frac{3(n+1)^2}{12})$

$=$ < Factor out $\frac{(n+1)}{12}$. >

$\frac{n^2(n+1)}{12}(6n - 4(2n+1) + 3(n+1))$

$=$ < Expand and simplify. >

$\frac{n^2(n+1)}{12}(n-1)$

$=$ < Multiply out $(n+1)(n-1)$. >

$\frac{n^2(n^2-1)}{12}$

QED.

This confirms the Main Result by hand proof. Now see if ACL2 confirms this result with the prover.

6 CS340d Laboratories

Laboratories (labs) are designed to enhance and deepen your understanding of specific material. During the course of this semester, there will be four labs. These labs build upon each other – so it's very important that you develop and submit the work identified for each lab.

The purpose of these laboratories is to make you familiar with the debugging process – by developing some of the basic debugging tools and then using them to debug some programs.

6.1 Lab 0

```
Laboratory 0
CS 340d
Unique Number: 52285
Spring, 2023

Given:  January 26, 2023
Due:  February 14, 2023
```

This laboratory concerns duplicating some of the functionality of the Unix "wc" command by writing an equivalent command in ACL2.

6.2 Lab 0 General Comments

Before we describe this homework assignment, we describe our philosophy for all laboratory assignments and for many of our homework assignments. We expect our programs to implement their requirements with mathematical precision, but program requirements are generally specified using natural language. To this point in your education, most programming assignments have included some description of what program you should write. Then, you are expected to *interpret* the requirements, specifications and other supplied documentation to produce a conforming result. It requires tremendous care and expert knowledge to write a precise description of any computation in a natural language – it is certainly beyond our ability to write such completely precise, natural-language specifications.

We would like to write mathematical specifications for our all of our programs. As a community of software developers, this approach would be extremely valuable, where it can be deployed, but it is not a mature discipline. Even so, we will attempt to use ACL2 where we can. For some programs, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level (program) commands. The running of these commands and their outputs will serve as executable simulators.

6.3 Lab 0 Requirements

This laboratory involves duplicating some of the functionality of the "wc" command using ACL2. Note, your program should also work on binary files.

Here, we include some remarks that might help you. The number of characters returned should be equal to the length of the file or input. The number of lines should be equal to the number of line-feed characters contained in the file. The number of words should be equal to the groups of characters separate by spaces, tabs, line feeds, and carriage returns. You should read the "wc" manual entry carefully.

But, the real specification of your ACL2-based "wc" is what the Linux version of "wc" does on the departmental Linux computers. This Linux program is your executable specification for this laboratory. Extra credit may be awarded if you find a discrepancy of some kind in the FreeBSD, Linux, or MacOS "wc" commands. What is a discrepancy? Absolutely any input file that caused your ACL2 version of "wc" to produce a correct result that is different than the UTCS Linux computers. Now, if you can argue to the class that even though your

implementation is inconsistent with the UTCS Linux result, that your result is correct – then you may have found a real bug! Bugs of this kind are always worth extra credit.

At the end of this laboratory assignment is some Lisp code that will help you get started – this code provides a template to help you start working.

Looking at the man page for the Linux wc program

The man page is a good place to start the quest for understanding the specification of the **wc** (word count) program. Here is a list, by no means exhaustive, of some of the functionality of **wc** that we need to support for completeness.

- Processing multiple files together, or working with standard input and output if no file names are provided.
- The functionality of **wc** is defined by its use of the unix command **iswspace(3)**. So this function should be studied and its functionality provided.
- **wc** supports cumulative counts for all files in one command.
- **wc** supports the following switches *-clmw*. The switches *-c* and *-m* are mutually exclusive.
- *-c* output the number of characters (bytes).
- *-l* output the number of lines.
- *-w* output the number of words.
- Accept input until EOF or ^D is received.
- Exits 0 on success, and > 0 if an error occurs.

To recreate the entire functioning of the Linux **wc** program would require additional work beyond what we are asking of you. We are only attempting to recreate the line, word, and character count results. Thus, our ACL2-based **wc** defines a subset of the Linux **wc** command in ACL2 Lisp.

6.4 Lab 0 Documentation

Finally, you need to include in your solution program a 50-line to 60-line comment as a Lisp-language comment that begins with a line containing only "#|" and ends with a line containing only "|#" that describes your ACL2 "wc" command. This description should be in the (approximate) format of a typical FreeBSD/Linux/MacOS manual entry. This description is a writing assignment associated with this laboratory – all of the laboratories in this class include a write-up of some kind. Remember, you are taking a class with a writing flag, and this kind of summary will be required for all of the class laboratory assignments. [Remark: I wonder how many ChatGPT submissions we will receive...]

6.5 Lab 0 Grading

Your laboratory will be graded with the follow weights:

```
70% - Functioning of your ACL2 "wc" implementation as specified above
30% - Written description of your "wc" command.
```

Be careful with what you write. We will be grading the functioning of your program on several hundred files. And, we will carefully read your documentation, looking for problems

(such as grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

6.6 Lab 0 Turn-in

Prior to the due date, we will post submission instructions.

6.7 Lab 0 Code Template

Below is a template you may use to complete your laboratory. Be sure to leave function **process-file** as your top-level function because our automated grading system will load your code and then call that function. Also, be sure that your entire code can be run without any intervention required; i.e., we should be able to load your file by: **(ld "<filename.lisp>")** .

```
; wc.lisp

; (ld "wc.lisp")

(in-package "ACL2")

; Some miscellaneous testing definitions.

(defmacro ! (x y)
  `(assign ,x ,y))

(defmacro !! (variable new-value)
  ;; Assign without printing the result.
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (if erp 'Error! ',variable))))


(defun count-words (list-of-chars)
  (declare (ignorable list-of-chars))
  "Count number of words."
  ;; Here is where you need to work!
  0)

(defun count-lines (list-of-chars)
  "Count number of LF characters."
  (if (atom list-of-chars)
      0
    (let ((char (car list-of-chars))
          (rest (cdr list-of-chars)))
      (if (equal char  #\Newline)
          (1+ (count-lines rest))
```

```
        (count-lines rest)))))

(defun check-for-ASCII-chars (list-of-chars)
  "Check that a list of characters are all ASCII characters."
  (if (atom list-of-chars)
      T
    (and (< (char-code (car list-of-chars)) 128)
         (check-for-ASCII-chars (cdr list-of-chars)))))

(set-state-ok t)

(defun process-file (filename state)
  "Reads and process the file <FILENAME>."
  (let ((str (read-file-into-string filename)))
    (if (not (stringp str))
        ;; If return value is not a string, we have an error condition.
        (mv (cw "Zero-length file.  Does the file ~p0 exist?~%" filename) state)

      (if (> (length str) 72057594037927936)
          ;; If LEN > 2^56, just give up!
          (mv NIL state)

        (let* ((list-of-chars (coerce str 'list))
               (ASCII-chars? (check-for-ASCII-chars list-of-chars)))

          (mv (list :ascii-characters ASCII-chars?
                    :number-of-characters (len list-of-chars)
                    :number-of-lines (count-lines list-of-chars)
                    :number-of-words (count-words list-of-chars)
                    )
              state))))))
```

6.8 Lab 1

```
Laboratory 1
CS 340d
Unique Number: 52285
Spring, 2023

Given:  February 16, 2023
Due:  March 7, 2023
```

This laboratory concerns counting the number of times each word appears in a document. This is an extension of our Lab 0 lab.

6.9 Lab 1 General Comments

Before we describe this laboratory assignment, we describe our philosophy for all laboratory assignments and for many of our homework assignments. We expect our programs to implement their requirements with mathematical precision, but program requirements are generally specified using natural language. To this point in your education, most programming assignments have included some description of what program you should write. Then, you are expected to *interpret* the requirements, specifications and other supplied documentation to produce a conforming result. It requires tremendous care and expert knowledge to write a precise description of any computation in a natural language – it is certainly beyond our ability to write such completely precise, natural-language specifications.

We would like to write mathematical specifications for our all of our programs. As a community of software developers, this approach would be extremely valuable, where it can be deployed, but it is not a mature discipline. Even so, we will attempt to use ACL2 where we can. For some programs, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level (program) commands. The running of these commands and their outputs will serve as executable simulators.

6.10 Lab 1 Requirements

This laboratory involves extending the functionality of Lab 0 by counting the number of times each word identified and counted in Lab 0 appears.

The expected result of running your new word-counting command should be an association list of words paired with the number of times each word appears.

At the end of this laboratory assignment is some Lisp code that will help you get started – this code provides a template to help you start working. This code also appears on the class website.

6.11 Lab 1 Documentation

Finally, you need to include in your solution program a 50-line to 60-line comment as a Lisp-language comment that begins with a line containing only `"#|"` and ends with a line containing only `"|#"` that describes your ACL2 `"wcnt"` command. This description should be in the (approximate) format of a typical FreeBSD/Linux/MacOS manual entry. This description is a writing assignment associated with this laboratory – all of the laboratories in

this class include a write-up of some kind. Remember, you are taking a class with a writing flag, and this kind of summary will be required for all of the class laboratory assignments.

6.12 Lab 1 Grading

Your laboratory will be graded with the follow weights:

```
70% - Functioning of your ACL2 "wcnt" implementation as specified above
30% - Written description of your "wcnt" command.
```

Be careful with what you write. We will be grading the functioning of your program on several hundred ASCII-only files. And, we will carefully read your documentation, looking for problems (such as grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

6.13 Lab 1 Turn-in

Prior to the due date, we will post submission instructions on Canvas.

6.14 Lab 1 Code Template

Below is a template you may use to complete your laboratory. Be sure to leave function **wcnt** as your top-level function because our automated grading system will load your code and then call that function. Also, be sure that your entire code can be run without any intervention required; i.e., we should be able to load your file by: **(ld "<filename.lisp>")** . We should be able to use your solution by evaluating: **(wcnt "filename" state)**.

```
; wcnt.lisp

; (ld "wcnt.lisp")
; (certify-book "wcnt" ?)
; (include-book "wcnt")

(in-package "ACL2")

; Some miscellaneous testing definitions.

(defmacro ! (x y)
  (declare (xargs :guard (symbolp x)))
  `(assign ,x ,y))

(defmacro !! (variable new-value)
  ;; Assign without printing the result.
  (declare (xargs :guard t))
  `(mv-let
    (erp result state)
    (assign ,variable ,new-value)
    (declare (ignore result))
    (value (if erp 'Error! ',variable))))
```

```
(defun wcnt (char-lst)
  "Count number of words."
  (declare (xargs :guard (character-listp char-lst))
           (ignorable char-lst))

  ;; Your solution should return an association list of words paired
  ;; with their frequency count.

  ;; Replace the constant result with your word counter function.

  '(("foo" . 3)
    ("bar" . 2)))


(defun check-for-ASCII-chars (char-lst)
  "Check that a list of characters are all ASCII characters."
  (declare (xargs :guard (character-listp char-lst)))
  (if (atom char-lst)
      T
    (and (< (char-code (car char-lst)) 128)
         (check-for-ASCII-chars (cdr char-lst)))))

(set-state-ok t)

(defun wcnt-file (filename state)
  "Reads and process the file <FILENAME>."
  (declare (xargs :guard (stringp filename)
                  :guard-hints
                  (("Goal"
                    :in-theory
                    (disable read-file-into-string2)))
                  :stobjs (state)))

  (let ((str (read-file-into-string filename)))
    (if (not (stringp str))
        ;; If return value is not a string, we have an error condition.
        (mv (cw "Zero-length file.  Does the file ~p0 exist?~%" filename) state)

      (if (> (length str) 72057594037927936)
          ;; If LEN > 2^56, just give up!
          (mv NIL state)

        (let* ((char-lst (coerce str 'list))
               (ASCII-chars? (check-for-ASCII-chars char-lst)))

          (mv (if ASCII-chars?
```

```
    (wcnt char-lst)
  "Non ASCII characters detected.")
state))))))
```

6.15 Lab 2

```
Laboratory 2
CS 340d
Unique Number: 52285
Spring, 2023

Given:  March 7, 2023
Due:  April 4, 2023
```

This laboratory concerns inserting, deleting, inspecting, and maintaining ordered lists and trees.

6.16 Lab 2 General Comments

Before we describe this laboratory assignment, we describe our philosophy for all laboratory assignments and for many of our homework assignments. We expect our programs to implement their requirements with mathematical precision, but program requirements are generally specified using natural language. To this point in your education, most programming assignments have included some description of what program you should write. Then, you are expected to *interpret* the requirements, specifications and other supplied documentation to produce a conforming result. It requires tremendous care and expert knowledge to write a precise description of any computation in a natural language – it is certainly beyond our ability to write such completely precise, natural-language specifications.

We would like to write mathematical specifications for our all of our programs. As a community of software developers, this approach would be extremely valuable, where it can be deployed, but it is not a mature discipline. Even so, we will attempt to use ACL2 where we can. For some programs, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level (program) commands. The running of these commands and their outputs will serve as executable simulators.

6.17 Lab 2 Requirements

This laboratory involves ordered sets of items, where the items are stored as lists and trees. We provide recognizers for both ordered lists and ordered trees. For both kinds of data structures, you are asked to write an insert, member, and delete function. We will provide the names and signatures of the functions you are to write.

At the end of this laboratory assignment is some ACL2 code that will help you get started – this code provides a template to help you start working. This code also appears on the class website.

6.18 Lab 2 Documentation

Finally, you need to include in your solution program a 80-line to 90-line comment as a Lisp-language comment that begins with a line containing only `"#|"` and ends with a line containing only `"|#"` that describes your ACL2 insert, member, and delete function for lists, and your ACL2 insert, member, and delete function for trees. This description should be in the (approximate) format of a typical FreeBSD/Linux/MacOS manual entry. This

description is a writing assignment associated with this laboratory – all of the laboratories in this class include a write-up of some kind. Remember, you are taking a class with a writing flag, and this kind of summary will be required for all of the class laboratory assignments.

6.19 Lab 2 Grading

Your laboratory will be graded with the follow weights:

```
70% - Functioning of your ACL2 member, insert, and delete functions
      for both data structures.
30% - Written description of your solutions.
```

Be careful with what you write. We will be grading the functioning of your program on several hundred ASCII-only files. And, we will carefully read your documentation, looking for problems (such as grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

6.20 Lab 2 Turn-in

Prior to the due date, we will post submission instructions on Canvas.

6.21 Lab 2 Code Template

Below is a template you should use to complete your laboratory. Be sure to leave the fully-defined functions as they are given here. Note, we will use an automated grading system will load your code and then call all of the functions you are expected to write. Be sure that your entire code can be run without any intervention required; i.e., we should be able to load your file by: **(ld "<filename.lisp>")** We should be able to use your solution by evaluating the various member, insert and delete functions repeatedly on a wide variety of data.

```
; Below, you need to provide the definitions for the LIST-based
; functions:

;          insrt (e x)
;          mbr (e x)
;          del (e x)

; And for the TREE-based functions:

;          bst-insrt (e x)
;          bst-mbr (e x)
;          bst-del (e x)


; Sets with Lists and Binary Trees       Scott, Vivek, & Warren

; (ld "lab2-mem-insrt-del.lisp" :ld-pre-eval-print t)
```

```
(in-package "ACL2")

(defun << (x y)
  "General less-than function."
  (declare (xargs :guard t))
  (and (lexorder x y)
       (not (equal x y))))

; Ordered sets represented as lists.

(defun setp (x)
  "Ordered list of objects."
  (declare (xargs :guard t))
  (if (atom x)
      (null x)
    (if (atom (cdr x))
        (and (atom (car x))
             (null (cdr x)))
      (let ((a (car x))
            (b (cadr x)))
        (and (atom a)
             (atom b)
             (<< a b)
             (setp (cdr x)))))))

(defun insrt (e x)
  "Insert E into ordered set X."
  (declare (xargs :guard (and (atom e)
                              (setp x))))
  ;; Replace X (below) with an Insert function body
  (list e x))

(defun mbr (e x)
  "Test whether E is in set X."
  (declare (xargs :guard (and (atom e)
                              (setp x))))
  ;; Replace form below with a Member function body
  (list e x))

(defun del (e x)
  "Delete element from set X, or do nothing if no E in X."
  ;; Observation: When E "larger" than (CAR x), we can stop
  (declare (xargs :guard (and (atom e)
                              (setp x))))
  ;; Replace form below with a Delete function body.
  (list e x))
```

```
; Ordered sets represented as trees.

(defun bstp (x)
  "Syntactic Tree Set Recognizer."
  (declare (xargs :guard t))
  ;; If NIL, x is the empty bst
  (if (atom x)
      (null x)
    (let ((obj (car x))
          (sbt (cdr x)))
      (and (atom obj)
           (consp sbt)
           (bstp (car sbt))
           (bstp (cdr sbt))))))

(defun tr<<e (x e)
  "All elements in X less than e."
  (declare (xargs :guard (bstp x)))
  (if (atom x)
      T
    (let ((obj (car x))
          (sbt (cdr x)))
      (and (<< obj e)
           (tr<<e (car sbt) e)
           (tr<<e (cdr sbt) e)))))

(defun e<<tr (e x)
  "All elements in X greater than e."
  (declare (xargs :guard (bstp x)))
  (if (atom x)
      T
    (let ((obj (car x))
          (sbt (cdr x)))
      (and (<< e obj)
           (e<<tr e (car sbt))
           (e<<tr e (cdr sbt))))))

(defun bst-ordp (x)
  "Recognizer for tree-based sets; all elements ordered."
  (declare (xargs :guard (bstp x)))
  (if (atom x)
      t
    (let* ((obj (car x))
           (sbt (cdr x))
           (lt (car sbt))
           (rt (cdr sbt)))
```

```
      ;; Consider both subtrees
      (and (bst-ordp lt)
           (bst-ordp rt)
           ;; Confirm that values "surround" OBJ
           (tr<<e lt obj)
           (e<<tr obj rt)))))

(defun bst-insrt (e x)
  "BST insert element"
  (declare (xargs :guard (and (atom e)
                              (bstp x)
                              (bst-ordp x))))
  ;; Replace form below with a tree-based Insert function body
  (list e x))

(defun bst-mbr (e x)
  "BST member, returns tree where e resides"
  (declare (xargs :guard (and (atom e)
                              (bstp x)
                              (bst-ordp x))))
  ;; Replace form below with a tree-based  Member function body
  (list e x))

(defun bst-del (e x)
  "BST delete, if element e present, delete it"
  (declare (xargs :guard (and (bstp x)
                              (bst-ordp x))))
  ;; Replace form below with a Delete function body.
  (list e x))
```

6.22 Lab 3

```
Laboratory 3
CS 340d
Unique Number: 52285
Spring, 2023

Given:  April 4, 2023
Due:  April 20, 2023
```

This laboratory concerns inserting, deleting, inspecting, and maintaining ordered lists and trees.

6.23 Lab 3 General Comments

Before we describe this laboratory assignment, we describe our philosophy for all laboratory assignments and for many of our homework assignments. We expect our programs to implement their requirements with mathematical precision, but program requirements are generally specified using natural language. To this point in your education, most programming assignments have included some description of what program you should write. Then, you are expected to *interpret* the requirements, specifications and other supplied documentation to produce a conforming result. It requires tremendous care and expert knowledge to write a precise description of any computation in a natural language – it is certainly beyond our ability to write such completely precise, natural-language specifications.

We would like to write mathematical specifications for our all of our programs. As a community of software developers, this approach would be extremely valuable, where it can be deployed, but it is not a mature discipline. Even so, we will attempt to use ACL2 where we can. For some programs, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level (program) commands. The running of these commands and their outputs will serve as executable simulators.

6.24 Lab 3 Requirements

This laboratory involves proving some theorems about the member, insert, and delete functions you defined in Lab 2. Below, you will find some theorems we ask you to prove about the functions you defined for your Lab 2 solution. We will also understand if you choose to re-define some of your Lab 2 solutions because you may find (when attempting to the prove properties requested below) that your specification functions may need improvement and/or correction.

At the end of this laboratory assignment there are some ACL2 DEFTHMs you are asked to prove. Note: the last request is very hard! Solutions for this last problem will be worth extra credit. The code (below) also appears on the class website.

6.25 Lab 3 Documentation

Finally, you need to include in your solution program a 80-line to 90-line comment as a Lisp-language comment that begins with a line containing only "#|" and ends with a line containing only "|#" that describes your approach to the proofs you are asked to perform.

This description should be in the (approximate) format of a typical FreeBSD/Linux/MacOS manual entry. This description is a writing assignment associated with this laboratory – all of the laboratories in this class include a write-up of some kind. Remember, you are taking a class with a writing flag, and this kind of summary will be required for all of the class laboratory assignments.

6.26 Lab 3 Grading

Your laboratory will be graded with the follow weights:

```
70% - For the proofs requested about your ACL2 member, insert, and
      delete functions for both data structures; this will be
      exhibited by the proofs you provide.

30% - Written description of your solutions.
```

Be careful with what you write. We will carefully read your documentation, looking for problems (such as grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

6.27 Lab 3 Turn-in

Prior to the due date, we will post submission instructions on Canvas.

6.28 Lab 3 Code Template

Below is a template you should use to complete your laboratory. Be sure to leave the fully-defined functions as they are given here. Note, we will use an automated grading system will load your code and then call all of the functions you are expected to write. Be sure that your entire code can be run without any intervention required; i.e., we should be able to load your file by: **(ld "<filename.lisp>")** We should be able to use your solution by evaluating the various member, insert and delete functions repeatedly on a wide variety of data.

```
; For your ACL2 definitions for the LIST-based functions:

;          insrt (e x)
;          mbr (e x)
;          del (e x)

; And your ACL2 TREE-based functions:

;          bst-insrt (e x)
;          bst-mbr (e x)
;          bst-del (e x)

; we ask you to prove the lemmas below.
```

```
; Sets with Lists and Binary Trees       Scott, Vivek, & Warren

; (ld "lab3-trees.lisp" :ld-pre-eval-print t)

(in-package "ACL2")

; Please prove these DEFTHMs about the list-based set functions
; you defined in Lab 2.  Please (re-)supply your definitions as
; you must submit a file that we can run with a single command
; (ld "<filename.lisp>").  Note, you may add helper definitions
; and lemmas.

(defthm setp-insrt
  ;; INSRT returns a SETP set.
  (implies (and (atom e)
                (setp x))
           (setp (insrt e x))))

(defthm mbr-insrt
  ;; E a member after its insertion.
  (implies (and (atom e)
                (setp x))
           (mbr e (insrt e x))))


(defthm mbr-a-mbr-insrt
  ;; A still a member after any insertion.
  (implies (and (atom a)
                (atom e)
                (setp x)
                (mbr a x))
           (mbr a (insrt e x))))

; +++++++++++++++++

(defthm setp-del
  ;; Deletion leaves a set.
  (implies (setp x)
           (setp (del e x))))

(defthm not-mbr-del
  ;; E not a member after deletion.
  (implies (and (atom e)
                (setp x))
           (not (mbr e (del e x)))))

; +++++++++++++++++
```

```
(defthm mbr-a-del-e
  ;; A still a member if different element deleted.
  (implies (and (atom a)
                (atom e)
                (not (equal a e))
                (setp x))
           (equal (mbr a (del e x))
                  (mbr a x))))

; +++++++++++++++++

; Please prove these DEFTHMs about the tree-based set functions you
; defined in Lab 2.

(defthm bstp-bst-insrt
  (implies (and (atom e)
                (bstp x))
           (bstp (bst-insrt e x))))

; Hint: two helper lemmas are needed.

(defthm bst-ordp-bst-insrt
  ;; BST-INSRT is ordered.
  (implies (and (atom e)
                (bstp x)
                (bst-ordp x))
           (bst-ordp (bst-insrt e x))))

; +++++++++++++++++

(defthm bst-mbr-bst-insrt
  ;; E a member after its insertion.
  (implies (and (atom e)
                (bstp x)
                (bst-ordp x))
           (bst-mbr e (bst-insrt e x))))

(defthm bst-mbr-a-mbr-insrt
  ;; A still a member after any additional insertion.
  (implies (and (atom a)
                (atom e)
                (bstp x)
                (bst-ordp x)
                (bst-mbr a x))
           (bst-mbr a (bst-insrt e x))))
```

```
; ++++++++++++++++

(defthm bstp-bst-del
  (implies (bstp x)
           (bstp (bst-del e x))))

; Hint: two helper lemmas are needed.

(defthm bst-ordp-bst-del
  (implies (and (bstp x)
                (bst-ordp x))
           (bst-ordp (bst-del e x))))



; EXTRA CREDIT Problem

; This is a hard problem!  If you succeed, you will receive extra
; credit after you explain your proof to the instructor or TA -- even
; though we can check it with ACL2.

(defthm bst-mbr-bst-del-e
  ;; Atom A still a member if different object (E) is deleted.
  (implies (and (atom a)
                (not (equal a e))
                (bstp x)
                (bst-ordp x))
           (equal (bst-mbr a (bst-del e x))
                  (bst-mbr a x))))
```

Doc Index

R

S

U

W