The CS340d Manual

Version 0.7.1
Updated May 5, 2021, Afternoon
See lectures: **Introduction to SMT**
See new homework: **Homework #9**

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 document contains information to help students navigate the UT CS340d course, Debugging and Verifying Programs. This information is arranged in a hierarchical manner. 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.

Our course introduces students to rigorous (sometimes formal) specification and (analytic) analysis techniques that should help them be better programmers and to analyze and understand methods 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.

1.1 Example Subjects and Problems

To give a feel for kinds of things we will investigate, we consider a few examples.

Addition by -1 (decrementing) and +1 (incrementing). Show invariant.

Termination of loop that counts bigger side until both sides are zero. Show termination argument.

Termination of 3n+1 problem.

Let's make this a bit more concrete. Consider the C-language subroutine below. What does this code do?

```
void does_what( int *x, int *y ) {

  *x = *x ^ *y;
  *y = *x ^ *y;
  *x = *x ^ *y;
}
```

What specification could we write for this code?

If we have a specification, how can we confirm that this code carries out our intention precisely?

Does this code need to be debugged?

Do we need a "test harness"? Can we analyze the code without using a C-language compiler?

Can this code be verified?

Will this code always terminate? Will this code always function correctly?

1.2 Authors

This document was created initially and is being maintained by Warren A. Hunt, Jr. Carl Kwan, Charles Sandel, and Scott Staley have provided significant additional content, and will be involved with our class this semester. Hunt expresses his heart-felt thanks for their contributions.

1.3 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.

Remember 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 structue or very large numbers of software components in a system.

In some 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 class. 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 will 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 requires students to write programs in 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.

Depending on the skills and interests of the students in this class, students may wish to develop a BDD 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. If time permits, we may learn the rudiments of Lisp – one of the oldest programming languages.

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 intetest 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.

1.4 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 sechedule 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.

NOTE! Because of the UT shutdown from Monday, February 15, 2021, through Tuesday, February 23, 2021, we missed week four (4) and the first half of week five (5). The balance of week 5 was spent getting back on track. Homework assignments were delayed two weeks. Lab #1 retained its due date, and additional information was provided to students so that they could complete it successfully.

Due to the UT shutdown, below we present a revised syllabus.

```
  Schedule Below is Approximate, Lectures Dates Will Change Slightly

  *** 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.
  *** NOTE:  Schedule changes will be announced in class and on Canvas

Week  Class    Date   Short Description

 0     00     Jan 19  Course Content Introduction,
                      Course Procedures and UT required disclosures
                      C-language Introduction
 0     01     Jan 21  C-language features: types, pointer, control flow
                      system calls, and the assert statement.
                      The simplest kind of verification -- co-simulation

 1     02     Jan 26  Lab 0, Discussion, Program model
                      y86 Assembler/Dis-assembler, Word count
 1     03     Jan 28  Compiling C to binary
                      How can correctness be stated?

 2     04     Feb  2  How to specify termination requirements?
                      Code annotation snippets
 2     05     Feb  4  How to specify correctness requirements?
                      Requirement statements, C assert

 3     06     Feb  9  Byte order, Bit count, Word count
 3     07     Feb 11  C-language loops, Program termination
                      Lab 0 Due; Lab 1, Discussion, Y86 Interpreter
```

```
 4     08     Feb 16  Ice storm:  No Class
 4     09     Feb 18  Ice storm:  No Class

 5     10     Feb 23  Ice storm:  No Class
 5     11     Feb 25  Picking up the pieces, discussion of Lab 1

 6     12     Mar  2  Array code, array-segment zero
 6     13     Mar  4  Specifying zero-filled array segment

 7     14     Mar  9  Discussion of Lab 1
 7     15     Mar 11  Discussion of the rest of this course.
                      Laboratory 1 due

        Mar 15-20  Spring Break

 8     16     Mar 23  Discussion of array copy (using ACL2)
                      Laboratory 2, the class GDB
 8     17     Mar 25  Discussion of insertion sort (using ACL2)
                      Laboratory 2, more GDB

 9     18     Mar 30  Mathematics for proof
 9     19     Apr  1  Automated verification tools, ACL2, Z3

10     20     Apr  6  Computer Arithmetic Analysis Practicum
10     21     Apr  8  Use of Z3 for program analysis, ACL2, Z3

11     22     Apr 13  Lab 3 assigned
11     23     Apr 15  Loop measures and termination proofs

12     22     Apr 20  Code analysis, Lab 2 due
12     23     Apr 22  Z3 advanced usage, ACL2, Z3
                      Lab 3, Debugging y86 binary programs

13     26     Apr 27  Binary-tree algorithms
13     27     Apr 29  Tree-based algorithms

14     28     May  4  Temporal Logic, Laboratory 3 due
14     29     May  6  Discussion of BDDs and BDD operations,
                      Stump the professor

15     30     May 12  Final Exam -- essentially a 3-hour class
              2-5 pm  Several quizzes will be given
```

1.5 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.

1.6 Homework

There will be ten to twelve homework assignments given during the semester. On most weeks, homework will be assigned on Thursdays and due twelve (12) days later (on Tuesday) 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!

1.7 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.

1.8 Quizzes

Over the course of the semester, there will twenty, or more, in-class quizzes. Quizzes are ten- to 20-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, but there will be a 3-hour, final-class session during the final exam period. During this final exam time, we will work on various problems and several quizzes will be given. The two lowest quiz grades will be dropped in calculating the final course grade.

1.9 Class Assessment

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

```
Component       Percentage of Course Grade
```

```
Quizzes:          30%  (any class period and final exam period)
Homework:         30%  (submitted to an on-line system)
Labs:             40%  (see individual weighting just below)
```

The Laboratory Projects will be weighted as follows:

```
Laboratory      Percentage of Course Grade
Lab 0:            10%
Lab 1:            10%
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.

The final exam (3-hour) period will similar to our in-semester classes, although it will be three hours and two or three quizzes will be administrated. At the time specified by the UT Registrar `http://registrar.utexas.edu/schedules/212/finals` we will hold a 3-hour (final) class. Note, the exact time for final exams is published by the UT Registrar – and we will use that time for our final. Each time we teach a large course, we are told by some student that he/she needs to take the exam early because of an existing airline reservation or whatever. Please make sure that your post-semester activities do not interfere with our final exam period.

1.10 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.

1.11 Electronic Class Delivery

This course will be offered electronically – using the Zoom video-conferencing system. We have adapted our course to this educational format, and this format requires a kind of cooperation and engagement that differs from typical classroom-based instruction. Thus, we have reduced the number and kind of interactions that are possible so as to reduce the amount of effort we have to spend on process. We will provide office hours multiple times each week so students may continue to engage with the Instructor and Teaching Assistant directly.

Note, we consider the security of the Zoom platform to be somewhat porous; that is, we do not have complete faith in electronic mechanisms for instruction. 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. And, if we are unable to cure your concern, please contact the UTCS Department Chairperson.

1.12 Code of Conduct

Guidelines for how we will conduct class will be announced as the semester progresses, but some things will differ from an in-person class.

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/2019-spring/cs340d/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/2019-spring/cs340d/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/2019-spring/cs340d/aug-2021.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/2019-spring/cs340d/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 1, 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
```

1.13 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.

1.14 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.

1.15 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.

1.16 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`).

1.17 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.

2 Lectures

Material for our in-class discussions and lectures will assist us in our cause: debugging and verifying programs.

The lectures are approximately in the order we will discuss them, but we will no doubt "jump" around as our class evolves.

2.1 Introduction to C

We will use C to implement a variety of algorithms, and we will use annotations to designate invariant program properties.

C-language Programming is (still) a mainstay of systems programming.

We use C because it is close "to the metal", and it is not so hard to see how it might be translated to assembly language and on to binary.

Sometimes, we will inspect the output of the compiler; that is, we will compile C-language code and specify that the compiler should generate x86 assembly code.

Although you may have been taught Java, C is language is most often used in system tools (e.g., compilers, operating systems, networking and so on).

We also use C in this course because it is likely that, if you have not used C in your work so far, you have used or are familiar with one or more of the following decendents or close cousins of C: C++, C#, C*, Java, Objective-C, Rust, and Swift.

2.2 Trivial C Example

Here is a "trivial" C program.

```
// A Very Simple C Program

#include "stdio.h"

int main()
{
  printf( "Hello!\n" );
}
```

How many files will be touched to compile this file?

How many system calls are necessary to compile this file?

Using a UTCS Linux machine, consider the output from:

```
strace   -e trace=all -f -o gcc-compile.log   gcc hello.c -o hello
```

Hopefully, this answer does not surprise you.

2.3 C Program Fragment

Debugging and verifying programs requires thoroughly understanding what code does.

Consider the following C-language program fragment.

```
void does_what( int *x, int *y ) {
```

```
    *x = *x ^ *y;
    *y = *x ^ *y;
    *x = *x ^ *y;
  }
```

What does "does_what" do? How could we confirm our conjecture?

2.4 Basic C Programming

Programming Process	**Tool Support**
Writing C Programs	Emacs, vi or other text editor
Compiling C Programs	cpp, and gcc
Linker, Loader, libs	gcc -o
Asssembler, Core image	gcc -a
Overall Process	Integrated Development Environment (IDE), e.g., XCODE, Emacs

Below is a sample C-language program. You will be asked to write and submit various C programs.

```
//  your-program-name.c

//  Created by <Student Name> on 1/19/2021.

// The C-preprocessor (cpp) expands #define directives

#define NULL          0
#define EXIT_KEY      '8'
#define MAXLEN        1000
#define square(x)     ((x) * (x))

// Unix Like systems provided include files
//   These file can be found in /usr/local/include
//   or /usr/include on some (classic) systems.

// When GCC is installed, headers can be found by:
// $ sudo find /usr -name stdio.h

#include <stdio.h>
#include <stdlib.h>
```

```
#include <string.h>

#include <ctype.h>          // Needed for isalpha(), isdigit(), etc.
#include <time.h>           // Needed for system call gettimeofday()
#include <sys/time.h>       // Needed for system call gettimeofday()

// You may include your own custom ‘‘*.h’’ files

#include ‘‘your-file-name.h’’

// C Main Program

int main(int argc, const char *argv[], char *envp[]) {
  // insert code here...

// Variable type definitions and initializations examples.

  int i = 0;
  float x;
  extern double val[];
  unsigned long int x, y, z ;

  (int *)func();  // func returns a pointer to an int.

// Type definitions and initializations for system call gettimeofday().
// Example from: @url{https://linuxhint.com/gettimeofday_c_language/}

  struct timeval tv;
  time_t t;
  struct tm *info;
  char buffer[64];

  gettimeofday(&tv, NULL);   //  Make the system call
  t = tv.tv_sec;

  info = localtime(&t);
  printf("%s",asctime (info));

  strftime (buffer, sizeof buffer, "Today is %A, %B %d.\n", info);
  printf("%s",buffer);

  strftime (buffer, sizeof buffer, "The time is %I:%M %p.\n", info);
  printf("%s",buffer);

  return 0;
}
```

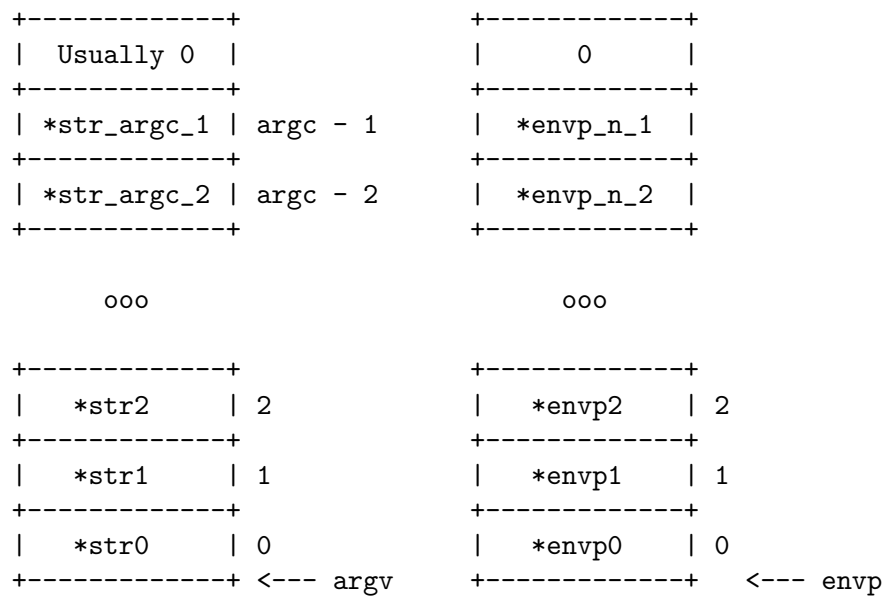
2.5 Unix Like Process

On systems that run FreeBSD/Linux/MacOS, the process is the primary resource that users are provided to access system resources.

The signature of the C-language starting point is:

```
int main( int argc, char *argv[], char *envp[] ) { ... }
```

What does this all mean? We consider the format of the input, shown below. For a quiz, we may ask that you write a C-langauge predicate that recognizes such a structure.



A user process, by way of the "main" routine in C, provides the number of command-line arguments, the arguments represented as strings, and the environment variables at process invocation.

What is the output signature? How can a return result be accessed?

On the web, there is a lot of information about environment variables; for example, see:

```
https://pubs.opengroup.org/onlinepubs/9699919799/
```

Below is a simple C program to look at the C sizeof operator, and to associate those sizes with the machine HW/SW.

```
//
//  sizeof-types.c
//  cs340d
//
//  Created by SMS on 1/23/2021.
//

#include <stdio.h>          // needed for printf function
#include <stdlib.h>
#include <errno.h>
#include <sys/utsname.h>    // needed for uname function
```

```
int main(int argc, const char *argv[], char *env[]) {
  //
  // insert code here...
  // Code from Stackoverflow for system call to uame:
  // https://stackoverflow.com/questions/3596310/c-how-to-use-the-function-uname
  //

  struct utsname buffer;

  errno = 0;
  if (uname(&buffer) != 0) {
    perror("uname");
    exit(EXIT_FAILURE);
  }

  printf("\nMachine information\n");
  printf("system name = %s\n", buffer.sysname);
  printf("node name   = %s\n", buffer.nodename);
  printf("release     = %s\n", buffer.release);
  printf("version     = %s\n", buffer.version);
  printf("machine     = %s\n", buffer.machine);

#ifdef _GNU_SOURCE
  printf("domain name = %s\n", buffer.domainname);
#endif

  //
  // Basic C types
  //
  printf("\nBasic C type-specifiers and storage allocation\n");
  printf("sizeof(%s) is: %lu bytes.\n", "void  ", sizeof(void));

  printf("sizeof(%s) is: %lu bytes.\n", "char ",  sizeof(char));
  printf("sizeof(%s) is: %lu bytes.\n", "char *",  sizeof(char *));
  printf("sizeof(%s) is: %lu bytes.\n", "short",  sizeof(short));
  printf("sizeof(%s) is: %lu bytes.\n", "int  ",  sizeof(int));
  printf("sizeof(%s) is: %lu bytes.\n", "long ",  sizeof(long));

  printf("sizeof(%s) is: %lu bytes.\n", "unsigned char ", sizeof(unsigned char));
  printf("sizeof(%s) is: %lu bytes.\n", "unsigned short", sizeof(unsigned short));
  printf("sizeof(%s) is: %lu bytes.\n", "unsigned int  ", sizeof(unsigned int));
  printf("sizeof(%s) is: %lu bytes.\n", "unsigned long ", sizeof(unsigned long));

  printf("sizeof(%s) is: %lu bytes.\n", "float ", sizeof(float));
  printf("sizeof(%s) is: %lu bytes.\n", "double", sizeof(double));
  printf("sizeof(%s) is: %lu bytes.\n", "long double", sizeof(long double));
  //
```

```
  // Some macOS (Darwin?) types
  //
  printf("\nmacOS (Darwin) type-specifiers and storage allocation\n");
  printf("sizeof(%s) is: %lu bytes.\n", "size_t" , sizeof(size_t));
  printf("sizeof(%s) is: %lu bytes.\n", "long long" , sizeof(long long));
  printf("sizeof(%s) is: %lu bytes.\n", "fpos_t" , sizeof(fpos_t));
  printf("sizeof(%s) is: %lu bytes.\n", "off_t" , sizeof(off_t));
  printf("\n");

  return EXIT_SUCCESS;

}
```

2.6 C Library Character

Here we list some of the C-library character recognizer functions. Such system supplied functions might be helpful for some of the C programming assignments in this course. But it would be more helpful for each student to write their own versions of these recognizer functions to become familiar with coding at this basic level.

```
#include <ctype.h>

int isalnum(int c);
int isalpha(int c);
int iscntrl(int c);
int isdigit(int c);
int isgraph(int c);
int islower(int c);
int isprint(int c);
int ispunct(int c);
int isspace(int c);
int isupper(int c);
int isxdigit(int c);
int isascii(int c);
int isblank(int c);
```

2.7 Locale Information

To gather information about the Locale being used, type

```
[hunt:~] % locale
LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL=
```

This is what I see on my Apple MacOS system.

As of January, 2021, 96.1% of Web traffic is in Unicode (UTF-8). More specific information can be found on the "w3techs.com" website.

https://w3techs.com/technologies/details/en-utf8/all/all

The website https://www.fileformat.info/infor/unicode/utf8.htm contains a short description of UTF-8, and it relation to Unicode.

For the rest of this lecture, I will assume that we are considering with files containing only ASCII characters.

2.8 Machine Language

Here we will discuss y86 machine-language programs.

2.9 Word Count Example

Here we will introduce the Unix "*wc*" command. wc is a simple command used to return information about the contents of a file.

On FreeBSD, Linux, and MacOS systems, one can type wc at the command line to get a count of the number of lines, words, and characters in a file.

```
%  wc
```

should print three natural numbers followed by a file name. This something like:

<lines> <words> <characters> <filename>

is printed for each <filename> processed (passed in as an command-line argument). If no <filename> is given, stdin is used by default.

The simplest thing reported by wc is the number of characters (really bytes) in a file. Generally, this is the same number of characters reported by the wc command – unless there is a Locale issue.

Some Locale setting recognize multi-byte characters; thus, the number of characters reported can be less than the number of bytes comprising a file, in this case.

One of the course laboratories will ask students to write a *"mywc"* program that functions very much like the widely available wc program.

2.10 Example Result of wc

When wc is run on the previous slide set, we get:

```
[hunt:~] % wc c-intro.org
     312    1039    7342 c-intro.org

[hunt:~] % ls -l c-intro.org
-rw-r--r--  1 hunt  hunt  7342 Jan 16 19:33 c-intro.org
```

This show 312 lines, 1039 words, and 7342 characters. This confirms that the number of characters reported is the same as the file length.

2.11 Number of Lines wc Reports

Given an ASCII file, the number of lines is exactly the number of ^J (0xa) characters (bytes) in a file.

In FreeBSD and Linux, the ^J is the line separator.

In MacOS X, ^J is the line separator.

Some Windows programs still use ^M^J as the line separator.

2.12 Number of Words wc Reports

This can be a bit subtle; it depends on what characters are considered to separate words.

Words are counted as character groups separated by white-space characters.

There are the "obvious" separators: <Beginning_of_File>, <End_of_File>, <TAB>, <SPACE>, and <LF>.

But, what about the other control characters? It appears that <CR> is also recognized as a word separator. Are there others?

2.13 Do Some Investigation

Before you attempt to implement your mywc program (for Laboratory 0), do some investigation.

What are the separator characters?

With your editor, can you create a file with all 128 ASCII characters?

In fact, can you create a file without a <LF> character?

2.14 Debugging and Verification

How can you compare the output of two programs?

Of course, you can run both programs and direct their output to different files, and then:

diff –brief wc-output.txt mywc-output.txt

Can you arrange that a large number of files can be tried and all of the results compared?

Can you write a program that takes two function pointers and a file descriptor (fd) and returns 1 if the two functions produce all the same outputs for identical input?

```
test-equiv( &f1(), &f2(), fd )  // where ‘‘fd’’ is a file descriptor
```

Does successful completion of the above mean the two functions (or program) are equivalent? If yes, why? If no, what is missing that is required for such a claim?

What is the difference between debugging and verification?

What is the difference between verification and validation?

2.15 Eliding Comments

In Laboratory 0, you are asked to implement the "-C" option that elides one-line, C-language comments before counting characters, lines, and words in a file.

The specification is given as: sed 's://.*$::g' | wc <options>

2.16 C Language Assertions

Here we describe a (partial) method to annotate C-language code with assertions. We will use the **assert** statement, which is actually a macro and may be removed/disabled at compile time using the **-DNDEBUG** flag.

Consider using the **tolower** library function. It's manual entry on my 2020 Macbook Air running MacOS 10.15.7 shows as:

```
TOLOWER(3)               BSD Library Functions Manual               TOLOWER(3)

NAME
     tolower, tolower_l -- upper case to lower case letter conversion

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <ctype.h>

     int
     tolower(int c);

     #include <ctype.h>
     #include <xlocale.h>

     int
     tolower_l(int c, locale_t loc);

DESCRIPTION
     The tolower() function converts an upper-case letter to the corresponding
     lower-case letter.  The argument must be representable as an unsigned char
     or the value of EOF.

     Although the tolower() function uses the current locale, the tolower_l()
     function may be passed a locale directly. See xlocale(3) for more informa-
     tion.

RETURN VALUES
     If the argument is an upper-case letter, the tolower() function returns the
     corresponding lower-case letter if there is one; otherwise, the argument is
     returned unchanged.

COMPATIBILITY
     The 4.4BSD extension of accepting arguments outside of the range of the
     unsigned char type in locales with large character sets is considered obso-
     lete and may not be supported in future releases.  The towlower() function
     should be used instead.
```

```
SEE ALSO
     ctype(3), islower(3), towlower(3), xlocale(3)

STANDARDS
     The tolower() function conforms to ISO/IEC 9899:1990 (‘‘ISO C90’’).

BSD                              July 17, 2005                              BSD
```

This library function replaces an upper-case character with a lower-case character. Given that this function concerns characters, we might wonder why it has a **int** input and output type.

```
int tolower(int c);
```

(Note, we do not discuss the location dependent argument.)

Normally, one would use it without thinking, such as:

```
int c, ch;

ch = tolower( c );
```

But, what should we know about arguments that can be provided to this library function? Why doesn't this function require a **char** type? Possibly, because of the second sentence which indicates that the input must be an **unsigned char** or **EOF**, which is generally defined as: **(-1)**.

So, as programmers, we should be suspicious, and program defensively. Let's modify (annotate) the code above to indicate what we expect to be true about the variable **c** upon entry. Let's confirm that the input is an ASCII character or **(-1)**.

```
int c, ch;

assert( isascii( c ) || c == EOF );

ch = tolower( c );
```

Is this good enough? What about an output condition? What should the output condition include?

```
int c, ch;

assert( isascii( c ) || c == EOF );

ch = tolower( c );

assert( isascii( ch ) || ch == EOF );  // OK, we want to know this

assert( islower( ch ) || ch == EOF );  // And, we want to know this as well

assert( ? );  // Can we say something about the correctness of ‘‘tolower’’?
```

Can we write a C-language specification that is (hopefully) different than the implementation that "speaks" to the correct functioning of the "islower" library function?

2.17 C Language Zero

What does it mean to zero a part of an array? For this class, it means that some number of bytes, starting at some particular address, are set to zero.

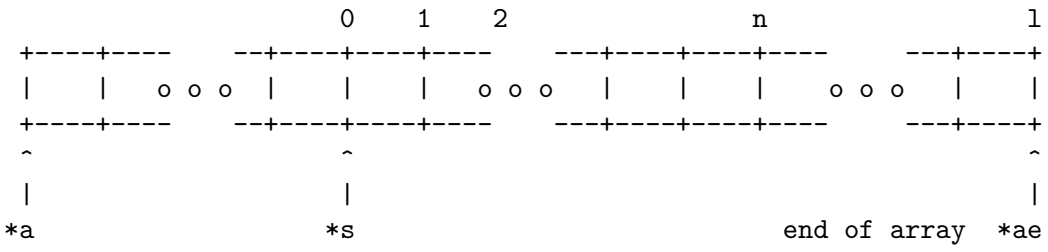
For our immediate discussion operations on arrays, we identify an array of bytes as starting at address "a" with length "l" bytes.

We want to assure that operations we perform are confined with in the bounds of the "a" array only.

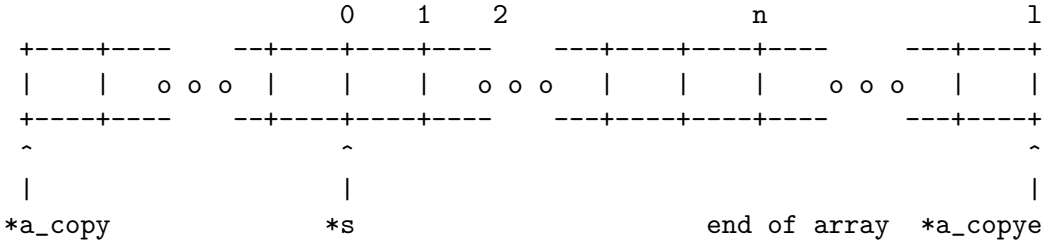
We should also be concerned that our array-modifying code does not "trash" the rest of memory; that is, the contents of other memory locations should not be changed.

Here is our prototypical array "a" of l bytes, where l is some positive number.

```
char a[l];
```




```
char a_copy[l];
```



Now, let us return to the issue of setting some number of bytes to zero.

We can start by creating code that uses a string library routine.

```
void my_zero1( void *s, size_t n ) {

  // Entrance predicate
  assert( 1 );  // Is a <= s && n >= 0 && s+n <= ae ?

  // Zero n bytes
  memset( s, 0, n );

  // Exit predicate
  assert( 1 );  // Check that the proper region was zero'ed.
}
```

Let's consider the **Entrance** and **Exit** predicates carefully. Reflecting on the (ASCII) diagram above, what do we need to know when we start? And, what do we know when we finish?

Is this enough? We need to know that **my_zero1** does not clear other bytes (locations). How could we specify that?

Next ZERO Routine

Now, let us consider the following code:

```
void my_zero2( void *s, size_t n ) {

  // Local copies of inputs

  size_t next_n = n;

  // Entrance predicate

  assert( n >= 0 && 1 );  // What else do we need to know?

  while( n != 0 ) {
    *s++ = 0;
    next_n = n-1;

    // Loop termination predicate

    assert( n > 0 && next_n >= 0 && next_n < n );

    n = next_n;
  }

  // Exit predicate

  assert( 1 );
}
```

We have three predicates:

- Entrance Predicate: What do we know when we start?
- Loop Termination Predicate: How do we know our routine will stop?
- Exit Predicate: What do we know when we finish?

Can **Entrance** and **Exit** predicates for the **my_zero2** routine be the same as the predicates for **my_zero1** routine? And, what is this **Loop Termination** predicate?

2.18 C Language Termination

A critical programming issue is to assure that all of our routines terminate. We will start out by being a bit informal

- straight-line code is expected to terminate; and
- every loop must be shown to terminate.

How can we know that straight-line code terminates? Straight-line code might call a library routine – what if the library code doesn't terminate?

We will assume that every subroutine we call has already been shown to terminate – but remember, subroutines, such as C-library code, may only terminate under very specific conditions.

How can we show that a subroutine will terminate? That is, what can we write to help assure ourselves that each subroutine terminates?

We will use a similar approach to that we have been using for our **Entrance** and **Exit** predicates; we will have a predicate that we will check each time we execute one "revolution" of a loop. But, what do we check?

We will define a **measure** that uses the variables involved in the termination of the loop we are checking to assure its termination.

Consider the termination of a C-language procedure that determines the length of a string.

```
size_t my_strlen( char *s ) {
  size_t len = 0;
  while( *s++ ) len++;
  return len;
}
```

What variable(s) should be involved in defining our measure?

How can we be sure that **my_strlen** will terminate?

2.19 C Language Casts

Consider the following program:

```
// check-addition.c                                Warren A. Hunt, Jr.

#include "stdio.h"
#include "stdlib.h"
#include "assert.h"

#define  UP_TO   (1 << 16)

int main( int argc, char *argv[], char *envp[] ) {

  int i, j;

  short x, y, sum;
  unsigned short ux, uy, usum;


  for( i = 0; i < UP_TO ; i++ ) {
```

```
    for( j = 0; j < UP_TO ; j++ ) {

      x  = (short)          i;
      ux = (unsigned short) i;

      y  = (short)          j;
      uy = (unsigned short) j;

      sum  = (short)          (  x +  y );
      usum = (unsigned short) ( ux + uy );

      if( sum != (short) usum )           // Comparison
        // Why do we need the cast above?
        {

          printf( "  i: %d,    j: %d, ",   i,    j );
          printf( "  x: %d,   ux: %d, ",   x  , ux );
          printf( "  y: %d,   uy: %d, ",   y,   uy );
          printf( "sum: %d, usum: %d." , sum, usum );
          printf( "\n" );
        }
    }
  }

  return( 0 );
}
```

Why do we need a C-language cast in the "Comparison" test (above)?

2.20 C Langauge-Like Copy

When I was working on Unix V7 (in about 1982), we made some performance benchmarks to determine where the V7 operating-system kernel spent its time. Even though the V7 kernel was providing all manner of services (opening files, starting processes, sending E-mail, etc.), the V7 kernel spent around 40% of its overall time copying data from one part of memory to another! Copying is something that computers do often, so it is important that it works efficiently and correctly.

On March 11, 2021, we discussed the focus of the class for the balance of the semester. Students wanted to see more verification technology, so we will start with showing ACL2-based specifications for initializing part of an array to a particular value.

Below is the definition of an array of 60-bit integers

```
(defstobj st

  (m :type (array (signed-byte 60)  ; array of 60-bit integers
                  (*init-m-size*))  ; with this initial length
     ;; Rational-number operations are not nearly as efficient as
     ;; integer operations, but for now this simplifies programming.
```

```
    :initially 0
    :resizable t)

  :inline t                          ; for performance
  :non-memoizable t                  ; also for performance
  :renaming                          ; for brevity
  ((update-mi  !mi) (m-length  ml)
   ))
```

The take away for the form above is that an array of 60-bit signed integers has been defined. For model execution efficiency, we define designate that, when used, the access function **mi** and the update function **!mi** should be "in-lined". We do not allow these functions to be memoized, and we rename the default names for update (from **update-mi** to **!mi**) and length (from **m-length** to **ml**).

The operator **mi** reads memory **m**, and is used as: **(mi addr st)**.

The operator **!mi** write memory **m**, and is used as: **(!mi addr value st)**.

Below is a definition for producing a list containing as many **i** values as there are element in **x**.

We will define list-based functions to serve as specification for our memory-based operations. First, we consider a pictorial diagram of what that might mean.

```
; We demonstrate the correctness of the memory-based initialization
; procedure INIT-IN-M by showing that when we map the range we wish
; to initialize ‘‘up to’’ a list, we find only initialized elements.

; But, that is not sufficient; we must also show that all other memory
; locations remain unchanged.

;                        O     <--- pair (tree node)
; left (the CAR) --->  / \
;                     /   \    <--- right (in Lisp, CDR)
; first item --->    a    O     <--- next pair
;                        / \
;                       /   \     and so on...
;  second item --->    b    O
;                               o o o        O   <--- another pair
;                                           / \
;  Function M-TO-L                         /   \
;  projects array (below)                 y     O   <--- last pair
;  to list (right, above)                     / \
;                                            /   \      conventional
;                             last item --->   z   NIL   <--- end

;  +-----+--    ---+-----+-----+--    ---+-----+-----+-----+--    ---+-----+
;  |     |  o o o  |  a  |  b  |  o o o  |  y  |  z  |     |  o o o  |     |
;  +-----+--    ---+-----+-----+--    ---+-----+-----+-----+--    ---+-----+
```

```
;  0              l                                  r                  end
;
;  Items to init:  <--------------------------------->
;
; Array occupies locations from l (inclusive) to r (exclusive).
```

The definitions and lemmas that follow attempt to capture the diagram above.

```
(defun init-lst (x i)
  "Make list of length x with i."
  (declare (xargs :guard t))
  (if (atom x)
      nil
    (cons i
          (init-lst (cdr x) i))))
```

Here is a function that recognizes that all entries in a list are **i**.

```
(defun all-i (x i)
  "All elements are equal to i."
  (declare (xargs :guard t))
  (if (atom x)
      t
    (and (equal (car x) i)
         (all-i (cdr x) i))))
```

Two properties of our **init-lst** function.

```
(defthm all-i-zero-lst
  ;; After INIT-LST, all elements are i.
  (all-i (init-lst x i) i))

(defthm len-zero-lst
  ;; Confirm length of initialize list.
  (equal (len (init-lst x i))
         (len x)))
```

The definition of setting each element of array **st** to **v** from pointer **l** up to **r**.

```
(defun init-in-m-simple (st l r v)
  "In-place sub-array initialization."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (<= l r)
                              (<= r (ml st))
                              (i60p v))
                  :stobjs st
                  :measure (nfix (- r l))))
  (if (zp (- r l))
      st
    (let ((st (!mi l v st)))
      (init-in-m-simple st (1+ l) r v))))
```

A provable equivalent definition with a limit on the array length.

```
(defun init-in-m (st l r v)
  "In-place sub-array initialization."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (<= l r)
                              (<= r (ml st))
                              (<= (ml st) *max-i60*)
                              (i60p v))
                  :stobjs st
                  :measure (nfix (- r l))))
  (mbe :logic
       (if (zp (- r l))
           st
         (let ((st (!mi l v st)))
           (init-in-m st (1+ l) r v)))
       :exec
       (let ((l (u60 l))
             (r (u60 r))
             (v (s60 v)))
         (if (= l r)
             st
           (let ((st (!mi l v st)))
             (init-in-m st (u60 (1+ l)) r v))))))
```

Below we show three lemmas about our initialization procedure.

```
(defthm mi-in-m-below-or-above--init-in-m
  ;; Array unchanged outside of initialized range
  (implies (and (natp l)
                (natp n)
                (or (< n l)    ;; below
                    (<= r n))) ;; above
           (equal (mi n (init-in-m st l r v))
                  (mi n st))))

(defthm mi-in-m-within--init-in-m
  ;; Within initialization range, each array value initialized.
  (implies (and (natp l)
                (natp r)
                (natp n) ;; l <= n < r
                (<= l n)
                (< n r))
           (equal (mi n (init-in-m st l r v))
                  v)))

(defthm m-seq-all-init
  ;; Within initialization range, all locations initialized.
```

```
  (implies (and (natp l)
                (natp r))
           (all-i (m-to-l (init-in-m st l r v)
                          l r)
                  v)))
```

As the C-programming language does not have a clear semantics, we would be forced to consider the binary produced by a C-language compiler. Thus, we would have to consider whether C-langauge programs left the memory as specified above.

Below is a pointer-based version of the C-language library function **memcpy**. NOTE: this program is not proved to work correctly when the target range overlaps the source range.

```
(defun memcpy (st l r p)
  (declare (xargs :guard (and (natp l)   ;; source start
                              (natp r)   ;; source end (exclusive)
                              (natp p)   ;; target start
                              (<= (ml st) *max-i60*)
                              (<= l r)
                              (<= r (ml st))
                              (let ((pr (+ p (- r l))))
                                (or (<= pr l) ; less
                                    (and (<= r p) ; greater
                                         (<= pr (ml st))))))
                  :stobjs st
                  :measure (nfix (- r l))))
  (mbe :logic
       (if (zp (nfix (- r l)))
           st
         (let ((l+1 (1+ l))
               (p+1 (1+ p))
               (e (mi l st)))
           (let ((st (!mi p e st)))
             (memcpy st l+1 r p+1))))
       :exec
       (if (= r l)
           st
         (let ((l+1 (u60 (1+ l)))
               (p+1 (u60 (1+ p)))
               (e   (s60 (mi l st))))
           (let ((st (!mi p e st)))
             (memcpy st l+1 r p+1))))))

(defthm stp-memcpy
  (implies (and (stp st)
                (natp l)       ;; source
                (natp p)       ;; target
                (<= r (ml st))
                (<= (+ p (- r l)) (ml st)))
```

```
           (stp (memcpy st l r p))))

(defthm ml-memcpy
  (implies (and (stp st)
                (natp l)       ;; source
                (natp p)       ;; target
                (<= r (ml st))
                (<= (+ p (- r l)) (ml st)))
           (equal (ml (memcpy st l r p))
                  (ml st))))

(defthm memcpy-different-from-target
  (implies (and (stp st)
                (natp l)
                (natp r)
                (natp p)
                (natp i)
                (<= r (ml st))
                (let ((pr (+ p (- r l))))
                  (or
                   (and (< i p)            ;; i below target
                        (<= pr (ml st)))
                   (and (<= pr i)          ;; i above target
                        (<= i (ml st))))))) ;; Why (= i (ml st)) ?
           (equal (mi i (memcpy st l r p))
                  (mi i st))))

(defthm m-to-l-help
   (implies (and (stp st)
                 (natp l)
                 (natp r)
                 (natp p)

                 ;; Source in range
                 (<= l r)
                 (<= r (ml st))
                 ;; (< r (ml st))

                 ;; Target in range
                 ;; (<= (+ p (- r l)) (ml st))
                 (< (+ p (- r l)) (ml st))

                 (or (< p l)    ;; Target below source
                     ;; (= p l)    ;; Target same doesn't work!  Why?
                     (<= r p))  ;; Target completely above source
                 (i60p v))
         (equal (m-to-l (!mi p v st) l r)
```

```
               (m-to-l st l r))))

(defthm memcpy-in-target-range
  (implies (and (stp st)
                (natp l)
                (natp r)
                (natp p)

                (<= l r)
                (< r (ml st))
                (< (+ p (- r l)) (ml st))

                (or (<  p l)
                    (=  p l)
                    (<= r p)))
           (equal (m-to-l (memcpy st l r p) p (+ p (- r l)))
                  (m-to-l st l r)))
  :hints
  ;; Delicate -- hint so ACL2 will use the MEMCPY induction machine.
  (("Goal" :induct (memcpy st l r p))))
```

2.21 C Langauge-Like Insertion Sort

Below is a set of ACL2 events that concern the correctness of a pointer-based, insertion-sort routine. This is provided so you can see what it takes to mechanically check the correctness of a pointer-based program.

```
;  isort.lisp                                  Warren A. Hunt, Jr.

; (ld "isort.lisp")
; (certify-book "isort" ?)
; (include-book "isort")


(in-package "ACL2")

; (include-book "std/util/bstar"          :dir :system)
; (include-book "std/testing/assert-bang" :dir :system)

(include-book "misc-events")
(include-book "limits")
(include-book "list-nth")
(include-book "lists")
(include-book "i60-listp")
(include-book "array")
```

```
; Definition of ordered

(defun orderedp (x)
  (declare (xargs :guard t))
  (if (atom x)
      t
    (if (atom (cdr x))
        t
      (and (lexorder (car x) (cadr x))
           (orderedp (cdr x))))))


; List-based insertion sort

(defun insert (e x)
  (declare (xargs :guard t))
  (if (atom x)
      (list e)
    (if (lexorder e (car x))
        (cons e x)
      (cons (car x)
            (insert e (cdr x))))))

(defun isort (x)
  (declare (xargs :guard t))
  (if (atom x)
      nil
    (if (atom (cdr x))
        x
      (insert (car x)
              (isort (cdr x))))))

; (assert! (equal (isort '(9 8 7 6 5 4 3 2 1 0)) '(0 1 2 3 4 5 6 7 8 9)))


; Insertion sort properties

(defthm orderedp-insert
  ;; Insertion leaves result ordered
  (implies (orderedp x)
           (orderedp (insert e x))))

(defthm how-many-insert
  ;; Insertion increased number of e2 elements
  (equal (how-many e (insert e2 x))
         (if (equal e e2)
             (1+ (how-many e x))
```

```
           (how-many e x)))
  :hints
  (("Goal" :induct (insert e2 x))))

(defthm len-insert
  ;; Insertion increased number of items by 1
  (equal (len (insert e x))
         (1+ (len x))))


(defthm orderedp-isort
  ;; ISORT returns ordered result
  (orderedp (isort x))
  :hints
  (("Goal" :in-theory (disable orderedp-insert))))

(defthm how-many-isort
  ;; Number of each element unchanged
  (equal (how-many e (isort x))
         (how-many e x)))

(defthm len-isort
  ;; Number of items unchanged
  (equal (len (isort x))
         (len x)))

(encapsulate
  ;; Redundant, because of the HOW-MANY theorem above
  ()
  (local
   (defthm perm-insert
     (implies (perm x1 x2)
              (perm (insert e x1)
                    (cons e x2)))))

  (defthm perm-isort
    ;; ISORT returns a permutation of its input
    (perm (isort x) x)))


; Relationship of LEXORDER to <

(defthm lexorder-is-<-when-m
  (implies (and (rationalp a)
                (rationalp b))
           (equal (lexorder a b)
                  (not (< b a))))
```

```
  :hints
  (("Goal" :in-theory (e/d (lexorder alphorder)())))


; Array occupies locations from l (inclusive) to r (exclusive).
;
;  +-----+--    ---+-----+-----+-----+-----+-----+--    ---+-----+
;  |     |  o o o  |     |     |     |     |     |  o o o  |     |
;  +-----+--    ---+-----+-----+-----+-----+-----+--    ---+-----+
;  0               l                       r                   end
;
;  Items to sort:  <---------------------->


(defun insert-e-in-m (st l r e)
  "Insert e into memory."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (< l r)
                              (<= r (ml st))
                              (< (ml st) *max-i60*)
                              (i60p e))
                  :stobjs st
                  :verify-guards nil
                  :measure (nfix (- r l))))
    (mbe :logic
         (if (zp (- r l))
             ;; Zero length array; do nothing
             st
           (let ((l+1 (1+ l)))
             (if (= l+1 r)
                 ;; Single-element array, perform insertion
                 (!mi l e st)
               (let ((nx-e (mi l+1 st)))
                 ;; Compare e with second element
                 (if (lexorder e nx-e)
                     ;; Write e if less than
                     (!mi l e st)
                   ;; Otherwise, m[l] <- m[l+1], and move on...
                   (let ((st (!mi l nx-e st)))
                     (insert-e-in-m st l+1 r e)))))))
         :exec
         (let ((l (u60 l))
               (r (u60 r))
               (e (s60 e)))
           (let ((l+1 (u60 (1+ l))))
             (if (= l+1 r)
```

```
                ;; Single-element array, perform insertion
                (!mi l e st)
              (let ((nx-e (s60 (mi l+1 st))))
                ;; Compare e with second element
                (if (<= e nx-e)
                    ;; Write e if less than
                    (!mi l e st)
                  ;; Otherwise, m[l] <- m[l+1], and move on...
                  (let ((st (!mi l nx-e st)))
                    (insert-e-in-m st l+1 r e)))))))))

(defthm stp-insert-e-in-m
  ;; State invariant maintained
  (implies (and (stp st)
                (natp l)
                (<= r (ml st))
                (i60p e))
           (stp (insert-e-in-m st l r e))))

(defthm ml-insert-e-in-m
  ;; Array size steady
  (implies (and (stp st)
                (natp l)
                (<= r (ml st)))
           (equal (ml (insert-e-in-m st l r e))
                  (ml st))))

(defthm mi-insert-e-in-m-less-than-l-is-mi
  ;; Insertion outside of range leaves array unchanged.
  (implies (and (stp st)
                (natp l)
                (natp i)
                (<= r (ml st))
                (or (< i l)
                    (and (<= r i)
                         (<= i (ml st)) ;; Why = ?
                         )))
           (equal (mi i (insert-e-in-m st l r e))
                  (mi i st))))

(verify-guards insert-e-in-m)


; Properties

(defthm m-to-l-!mi-above
  ;; Inductive observation
```

```
  (implies (and (stp st)
                (natp l)
                (natp l+)
                (< l l+))
           (equal (m-to-l (!mi l e st) l+ r)
                  (m-to-l st l+ r))))

(defthm cons-is-same-as-insert-when-e-less-than-m-{l+1}
  ;; Obvious consequence -- stupid theorem-prover trick
  (implies (and (stp st)
                (natp l)
                (<= r (ml st))
                (i60p e)
                (<= e (mi l st)))
           (equal (cons e (m-to-l st l r))
                  (insert e (m-to-l st l r))))
  :hints
  (("Goal"
    :do-not-induct t
    :expand (m-to-l st l r))))

(defthm insert-e-in-m-correctness
  ;; Relatinoship between INSERT and memory-based insertion
  (implies (and (stp st)
                (natp l)
                (natp r)
                (< l r)
                (<= r (ml st))
                (i60p e))
           (equal (m-to-l (insert-e-in-m st l r e) l r)
                  (insert e (m-to-l st (1+ l) r))))
  :hints
  (("Goal"
    :induct (insert-e-in-m st l r e))))

(in-theory (disable insert-e-in-m))


(defun isort-in-m (st l r)
  "ISORT insertion iteration."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (< l r)
                              (<= r (ml st))
                              (< (ml st) *max-i60*))
                  :verify-guards nil
                  :stobjs st
```

```
                  :measure (nfix (- r l))))
  (mbe :logic
       (if (zp (- r l))
           st
         (let ((l+1 (1+ l)))
           (if (= l+1 r)
               ;; One-element array; do nothing
               st
             ;; Sort rest (tail) of array; then insert first element
             (let ((st (isort-in-m st l+1 r)))
               (if (mbt (stp st))
                   (insert-e-in-m st l r (mi l st))
                 st)))))
       :exec
       (let ((l (u60 l))
             (r (u60 r)))
         (let ((l+1 (u60 (1+ l))))
           (if (= l+1 r)
               ;; One-element array; do nothing
               st
             ;; Sort rest (tail) of array; then insert first element
             (let ((st (isort-in-m st l+1 r)))
               (insert-e-in-m st l r (mi l st))))))))

(defthm ml-isort-in-m
  ;; Array size steady
  (implies (and (stp st)
                (natp l)
                (<= r (ml st)))
           (equal (ml (isort-in-m st l r))
                  (ml st))))

(defthm stp-isort-in-m
  ;; State invariant maintained
  (implies (and (stp st)
                (natp l)
                (natp r)
                (<= l r)
                (<= r (ml st)))
           (stp (isort-in-m st l r))))

(defthm mi-isort-in-m-less-than-l-is-mi-help
  ;; Insertion outside of range leaves array unchanged.
  (implies (and (stp st)
                (natp l)
                (natp l+)
                (< l l+)
```

```
               (<= r (ml st)))
           (equal (mi l (isort-in-m st l+ r))
                  (mi l st))))

(defthm mi-isort-in-m-less-than-l-is-mi
   (implies (and (stp st)
                 (natp l)
                 (natp i)
                 (natp r)
                 (<= r (ml st))
                 (or (< i l)
                     (and (<= r i)
                          (< i (ml st)))))
            (equal (mi i (isort-in-m st l r))
                   (mi i st))))

(verify-guards isort-in-m)


(defthm mi-isort-in-m-in-range
  ;; All values in memory are OK.
  (implies (and (stp st)
                (natp l)
                (natp r)
                (natp i)
                (<= l r)
                (<= r (ml st))
                (< i (ml st)))
           (and (<= *min-i60* (mi i (isort-in-m st l r)))
                (<= (mi i (isort-in-m st l r)) *max-i60*)))
  :rule-classes (:linear :rewrite))



; Correctness

;                       O
;                      / \
;                     /   \
;                    a    O
;                        / \
;                       /   \
;                      b    O
;                                  o o o        O
;                                              / \
;                                             /   \
;  Map array (below)                         y     O
```

```
;  to list (right, above)                        / \
;                                                /   \
;                                               z   NIL

;  +-----+--    ---+-----+-----+--    ---+-----+-----+-----+--    ---+-----+
;  |     |  o o o  |  a  |  b  |  o o o  |  y  |  z  |     |  o o o  |     |
;  +-----+--    ---+-----+-----+--    ---+-----+-----+-----+--    ---+-----+
;  0               l                                 r                  end
;
;  Items to sort:  <--------------------------------->
;
; Array occupies locations from l (inclusive) to r (exclusive).


(defthm insort-m-correctness
  (implies (and (stp st)
                (natp l)
                (natp r)
                (<= r (ml st))
                (<    l    r)
                (i60p e))
           (equal (m-to-l (isort-in-m st l r) l r)
                  (isort (m-to-l st l r)))))

(in-theory (disable isort-in-m))

; isort Correctness theorem




; To run some examples

(defun init-mem (st l r n)
  "Initialize memory with acending sequence."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (<= l r)
                              (<= r (ml st))
                              (< (ml st) *max-i60*)
                              (integerp n)
                              (and (<= *min-i60* n)
                                   (<= (+ n (- r l)) *max-i60*)))
                  :stobjs st
                  :measure (nfix (- r l))))
  (mbe :logic
```

```
       (if (zp (- r l))
           st
         (let ((st (!mi l n st)))
           (init-mem st (1+ l) r (1+ n))))
       :exec
       (let ((l (u60 l))
             (r (u60 r)))
         (if (= r l)
             st
           (let ((st (!mi l n st)))
             (init-mem st (u60 (1+ l)) r (s60 (1+ n)))))))))


(defun init-mem-reverse (st l r n)
  "Initialize memory with decending sequence."
  (declare (xargs :guard (and (natp l)
                              (natp r)
                              (<= l r)
                              (<= r (ml st))
                              (< (ml st) *max-i60*)
                              (integerp n)
                              (and (<= *min-i60* (- n (- r l)))
                                   (<= n *max-i60*)))
                  :stobjs st
                  :measure (nfix (- r l))))
  (mbe :logic
       (if (zp (- r l))
           st
         (let ((st (!mi l n st)))
           (init-mem-reverse st (1+ l) r (1- n))))
       :exec
       (let ((l (u60 l))
             (r (u60 r)))
         (if (= r l)
             st
           (let ((st (!mi l n st)))
             (init-mem-reverse st (u60 (1+ l)) r (s60 (1- n)))))))))

; Some examples

#||
(resize-m 10 st)

(! m '(9 8 7 6 5 4 3 2 1 0))
(l-to-m st 0 (@ m))
(m-to-l st 0 10)
```

```
(isort-in-m st 3 7)  ; Sort from postion 3 through 6.
(m-to-l st 0 10)


(resize-m 100000 st)

;; Sort in-order elements.
(init-mem st 0 100000 0)
(m-to-l st 0 10)
(time$ (len (isort (m-to-l st 0 100000))))
(time$ (isort-in-m st 0 100000))

;; Sort elements in reverse order.

(init-mem-reverse st 0 100000 100000)
(m-to-l st 0 10)
(time$ (len (isort (m-to-l st 0  20000))))
; (time$ (len (isort (m-to-l st 0 100000))))

(time$ (isort-in-m st 0  20000))
(time$ (isort-in-m st 0 100000))



(init-mem-reverse st 0 100000 100000)
(time$ (isort-in-m st 0 100000))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 26.08 seconds realtime, 26.00 seconds runtime
; (64 bytes allocated).


(init-mem-reverse st 0 100000 100000)
(time$ (len (isort (m-to-l st 0 100000))))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 300.49 seconds realtime, 301.02 seconds runtime
; (80,002,400,064 bytes allocated).
100000
ACL2 !>

||#
```

2.22 Review of Basic Logic

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 pleasent) memories:

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

2.22.1 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.

2.22.2 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. $p \wedge q$?

2.22.3 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.

2.22.4 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., $\equiv, \neg, \vee, \wedge, \mathit{true} \ldots$). 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.

2.22.5 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.

2.22.6 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.

2.22.6.1 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	

2.22.6.2 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** : $\frac{\mathbf{X=Y},\ \mathbf{Y=Z}}{\mathbf{X=Z}}$

2.22.6.3 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)

2.22.6.4 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 (36) $p\, U (p\, U q)$ (Left absorption of U)
- Prove (141) $p\, U [] p \equiv [] p$ (Absorption of U into $[]$)

2.22.6.5 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) $(p \Rightarrow q)\, W p$

2.22.6.6 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) $[]((p \vee [] q) \wedge ([] p \vee q)) \equiv [] p \vee [] q$

2.22.6.7 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

2.22.6.8 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.

2.22.6.9 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
```

2.23 Review of Linear Temporal Logic

In this section we provide a breif 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

2.23.1 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).

2.23.2 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 specificaton 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.*
- Well-formedness 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 infintely 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.

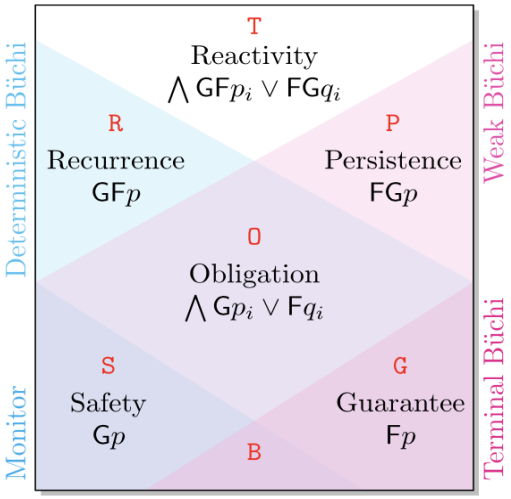
- **Recurrence**: these properties are the dual of Persistence. They are formulas of the form: $[] \diamond p$. They express the notion that the trace of p contains infinitely many p-positions. They are used in expressing properties of Justice and Fairness in LTL.

- **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 $\diamond [(x = 1) \wedge \diamond (y = 2)]$



2.23.3 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.

2.23.4 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*.

2.23.4.1 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 dfferent 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	...

2.23.4.2 Direct Proof

Ok, let's do some direct proofs.

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

2.23.4.3 Mutual Implication Proof

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

2.23.4.4 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: $[](]p \Rightarrow q) \vee [](]q \Rightarrow p)$

2.23.4.5 Proof by Contradiction

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

2.23.4.6 Proof by Contrapositive

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

2.23.4.7 Proof by Case Analysis

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

2.23.4.8 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:

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

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:

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

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.

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

2.23.5 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.

2.23.6 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)

2.24 Introduction to SMT

Consider a formula over a Boolean algebra, say,

$$a \wedge \neg a$$

or

$$a \wedge (\neg a \vee b \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (b \vee \neg d)$$

It is clear that the former formula will always evaluate to *false* independently of the truth assignments of a. But what about the latter formula? Does it have an assignment such that it will evaluate to *true*? This is the Boolean satisfiability (SAT) problem.

One naïve approach to SAT is to simply assign variables with arbitrary truth values until a satisfying assignment is found or if there are no other cases. Visually, this can be represented with a tree. For example, suppose the formula of interest is

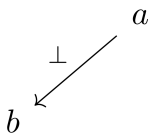
$$(a \vee b) \wedge (a \vee \neg b)$$

To see if this formula is SAT, start with a branching variable:

$$a$$

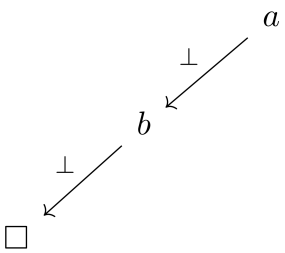
Original formula: $(a \vee b) \wedge (a \vee \neg b)$ Current formula: $(a \vee b) \wedge (a \vee \neg b)$

Pick an arbitrary value for the branching variable:



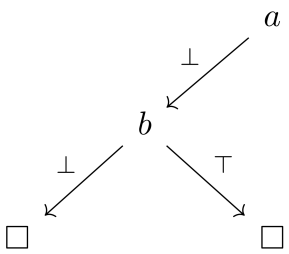
Original formula: $(a \vee b) \wedge (a \vee \neg b)$ Current formula: $(\bot \vee b) \wedge (\bot \vee \neg b)$

Pick an arbitrary value for another variable:



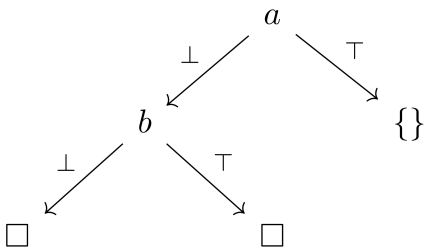
Original formula: $(a \vee b) \wedge (a \vee \neg b)$ Current formula: $(\bot \vee \bot) \wedge (\bot \vee \top)$

If a contradiction occurs, backtrack to the last variable and pick another value:



Original formula: $(a \vee b) \wedge (a \vee \neg b)$ Current formula: $(\bot \vee \top) \wedge (\bot \vee \bot)$

Repeat until sat (or until the search space has been exhausted):



Original formula: $(a \vee b) \wedge (a \vee \neg b)$ Current formula: $(\top \vee b) \wedge (\top \vee \neg b)$

Here, the box denotes an unsatisfactory assignment (unsat) and the empty clause $\{\}$ denotes a satisfactory assignment (sat). In the example, the initial choice of $a = \bot$ was a poor one in that the path to the solution is the longest possible. It is clear that this approach has exponential time complexity.

Now what if we replace the Boolean variables with formulas that evaluate to true or false? For example, if $a \equiv (x \leq 4)$ and $b \equiv (y \leq x)$, then our example from before would become

$$(a \vee b) \wedge (a \vee \neg b) \equiv [(x \leq 4) \vee (y \leq x)] \wedge [(x \leq 4) \vee \neg(y \leq x)].$$

Is this formula "sat"?

Let's try something trickier:

$$(x \geq 2) \wedge (y < 3) \wedge (x \leq y) \wedge (y \leq x)$$

Is this sat?

Maybe: what are x and y? If they are real (or even rational), then yes! Pick $x = 2.5$ and $y = 2.5$. If they are integers, then no!

More general formulas involving reals:

$$(x_1 + x_2 \leq x_3) \wedge \neg(x_3 + 0 \leq x_4) \wedge (x_4 \leq x_3 - 1) \wedge \cdots$$

Things to note:

1. The formula above only involves $+$, $-$, $\wedge$, $\neg$, $\leq$.

2. This is a *linear programming* problem.
3. Linear programming has both polynomial-time algorithms (e.g., interior point methods) and exponential-time algorithms (e.g., Simplex).
4. What about $\vee$? Use $\wedge$ and $\neg$.
5. What about $>$? Use $\leq$ and $\neg$.
6. What about $\geq$? Switch.
7. What about $=$? Use $\leq$ and $\geq$.

If we're dealing with a *theory of reals* over $+$, $-$, and $\leq$, then we have general methods for deciding whether the formula is sat.

If we're dealing with a *theory of integers* over the same symbols, then this is an *integer programming problem*, which is decidable, but NP-complete.

What if we include multiplication for the above theory of reals? Sure, but doubly-exponential.

What if we include multiplication for the above theory of integers? No: undecidable.

Note that throughout, we don't talk about quantifiers. Things would be even harder if we did.

What was the point of all this? Hopefully, by now you're convinced that there are theories in which we can decide whether any formula is satisfiable, hence, *satisfiability modulo theories* (SMT).

Why do you care? Some interesting theories for you:

- Theory of Arrays – decidable but NP-complete.
- Theory of Inductive Data Types – decidable and NP-complete, but usually "fast".
- Theory of Bit-Vectors
- Theory of Pointers and Reachability

You'll see in the homework how to use SMT solvers to help you verify your programs. And we'll go over some more examples later.

2.25 Z3 Examples

In this lecture, we're going to go through a couple examples in Z3. First, let's take the examples from our previous lecture and see how they look in Z3.

2.25.1 Z3 Booleans

Recall that the following formula was unsat:

$$a \wedge \neg a$$

In Z3:

```
## Initialise Boolean variables
a = Bool('a')
## Initialise a solver
s = Solver()
## Add formula to solver
```

```
s.add(And(a,Not(a)))
## Check if sat
print(s.check())
```

Now if our formula is unsat, then the negation of our formula should be sat.

```
## Initialise a new solver
s = Solver()
s.add(Not(And(a,Not(a))))
print(s.check())
print(s.model())
```

Great, but that was a pretty trivial example. How about that complicated looking one?

$$a \wedge (\neg a \vee b \vee \neg c) \wedge (\neg b \vee \neg c) \wedge (b \vee \neg d)$$

In Z3:

```
## Initialise Boolean variables
a = Bool('a')
b = Bool('b')
c = Bool('c')
d = Bool('d')
## terms
t1 = Or(Not(a), b, Not(c))
t2 = Or(Not(b), Not(c))
t3 = Or(Not(b), Not(d))
## putting everything into a single formula
f = And(a,t1,t2,t3)
## Initialise a solver
s = Solver()
s.add(f)
print(s.check())
print(s.model())
```

We can even see a satisfying assignment:

```
[b = False, a = True, c = False, d = False]
```

What happens if we check for the satisfying assignment of $\neg(a \wedge \neg a)$? Remember, solving a sat problem is different from proving the formula is true.

One more for good measure:

$$(a \vee b) \wedge (a \vee \neg b)$$

In Z3:

```
a = Bool('a')
b = Bool('b')
t1 = Or(a,b)
t2 = Or(a,Not(b))
f = And(t1,t2)
s = Solver()
s.add(f)
print(s.check())
print(s.model())
```

2.25.2 Z3 Integers

Let's take a look at how Z3 can solve integer programs. Recall that integer programming is NP-complete, whereas linear programming has polynomial-time algorithms. In the following examples, you shouldn't see much performance difference whether the programs are over the reals or integers, but this is because the our programs are simple. This is important to remember when you deal with larger programs in your day-to-day optimisation problems.

Consider the program that we translated from the previous sat problem:

$$(a \vee b) \wedge (a \vee \neg b) \equiv [(x \leq 4) \vee (y \leq x)] \wedge [(x \leq 4) \vee \neg(y \leq x)].$$

In Z3, this would look like

```
## Initialise x and y as integers
x = Int('x')
y = Int('y')
## Initialise solver and add program
s = Solver()
p = And(Or(x <= 4, y <= x), Or(x <= 4, Not(y <= x)))
s.add(p)
## Solve program
print(s.check())
print(s.model())
```

which returns

```
sat
[x = 4, y = 5]
```

Now let's take a look at something that might come up in an optimisation problem: Maximise y subject to

$$0 \leq x,$$
$$0 \leq y,$$
$$y - x \leq 1,$$
$$3x + 2y \leq 12,$$
$$2x + 3y \leq 12.$$

Well, we can check if $y \geq 2$ easily enough:

```
x = Int('x')
y = Int('y')
s = Solver()
## Can we satisfy y >= 2 given the constraints?
s.add(2 <= y)

s.add(0 <= x)
s.add(0 <= y)
s.add(-x+y <= 1)
s.add(3*x+2*y <= 12)
s.add(2*x+3*y <= 12)
```

```
print(s.check())
print(s.model())
```

Indeed:

```
sat
[y = 2, x = 1]
```

How about $y \geq 3$?

```
s.add(3 <= y)
print(s.check())
```

No:

```
unsat
```

So $y = 2$ is maximum value of y given the above constraints, which means we actually solved an *optimisation* integer program!

Remember that so far we've only dealt with *decision* problems. Usually, when people talk about linear and integer programming, they are referring to the optimisation problem. So if we want to find an *optimal* y using only the tools we've seen so far, then we would need solve the constraints above for multiple values. This isn't efficient and integer programming is hard to efficiently solve anyways, but what if we're dealing with reals, or if we're forced to deal with integers? Having to write out all these guesses for the desired optimised variable would be very annoying. Luckily, Z3 has a way to automate this optimisation for us:

```
## x, y are integers
x = Int('x')
y = Int('y')
## Initialise optimizer
o = Optimize()
## Maximise y subject to the following constraints
o.maximize(y)

o.add(0 <= x)
o.add(0 <= y)
o.add(-x+y <= 1)
o.add(3*x+2*y <= 12)
o.add(2*x+3*y <= 12)
print(o.check())
print(o.model())
```

As expected, Z3 returns:

```
sat
[y = 2, x = 1]
```

Some things to think about:

- Try the example programs we've seen so far on reals instead of integers.
- What if we do *mixed integer-linear* programming (i.e., programs where some variables are over the integers and some are over the reals)? How might this change the efficiency of solving programs?
- Think of some "real life" examples where you would use linear, integer, and mixed integer-linear programming to solve a problem.

- Think of some "real life" examples for sat solving.

Next, we'll look at some applications of SMT solving that are more relevant to your everyday activities as a computer scientist, engineer, or programmer.

2.26 SMT Applications

Work in progress. In the meantime, here is the cleaned-up code from lecture for verifying properties about 64-bit ceiling/floor functions.

```
from z3 import *

## We want to prove:
##  a. ceiling (n/2) = floor ((n + 1) /2)
##  b. floor (n/2) = ceiling ((n - 1) /2)
## e.g., floor(3/2) = floor(1.5) == 1 == 2/2 = ceiling((3-1)/2)
## (adapted from SAT/SMT by Example)

## Recall:
##  floor (1.2) == 1 (rounds down to the nearest integer)
##  ceil  (1.2) == 2 (rounds up to the nearest integer)

## "Block floating point"
## Numbers represented in hex, where the two left-most hex digits are those
## left of the "decimal", e.g.
##   in decimal:    99999999999999999999999.00
##   in hex:     0x ffffffffffffffffffffff 00

## Round down by making the lower 16 bits zero
def floor (x):
  return x & 0xffffffffffffff00

## If x is not an integer, round up, else it should stay the same
def ceiling (x):
            # check if x is an integer
  return If(( x & 0xffffffffffffff00 != x), # x & 0xff != 0   def 1
            # round down and add one if x is not an integer
            ( x & 0xffffffffffffff00 ) + 0x100 ,  # e.g. floor(1.2) -> floor(1.2) + 1 == 1
            # return x if x is already an integer
            x)

n = BitVec ('n', 64)
x = BitVec ('x', 64)
s = Solver()

## If n is an integer, then floor(n) == ceiling(n)
#s.add(Not(Implies(n & 0xffffffffffffff00 == n, floor(n) == ceiling(n))))
```

```
## Two ways to check if x is an integer:
##   def 1: x & 0xfffffffffffffff00 != x),
##   def 2: x & 0xff != 0
## Are they equivalent?

## This checks whether def 1 is equivalent to def 2
#s.add((x & 0xfffffffffffffff00 != x) != (x & 0xff != 0))

## Ensure n is always positive, no overflow
s.add(n < 0x8000000000000000 )
## This equivalent to
##  0b1000000000000000000000000000000000000000000000000000000000000000 (imagine 64 bits ... I didn't count)

## Finally, our proofs:

## Proof of a :
##   ceil(n/2) == floor( (n+1)/2) )
##   e.g. ceil(5/2) == ceil(2.5) == 3 == floor(6/2) == floor((5+1)/2)
#s.add( ceiling (n/2) != ( floor ((n+0x100)/2)))
formula1 = ( ceiling (n/2) == ( floor ((n+0x100)/2)))

## Proof of b :
##   floor(n/2) == ceil((n-1)/2) formula 2
##   e.g. floor(5/2) == floor(2.5) == 2 == ceil(4/2) == ceil((5-1)/2)
#s.add( floor (n/2) != ( ceiling ((n-0x100)/2)))
formula2 = floor (n/2) == ( ceiling ((n-0x100)/2))

formula = And(formula1, formula2)
s.add(Not(formula))

## Show terms in the Solver
print(s)
## Check if our theorems are true.
print (s.check()) # unsat


## BE CAREFUL!
## Why did I add Not(And(formula1,formula2)) to the solver?

## If you want to prove both formulas a and b, you cannot just
##   solver.add(Not(a))
##   solver.add(Not(b))
##   solver.check()
## This will prompt Z3 to find a satisfying assignment to ~a and ~b .
## If Z3 returns unsat, then you will have proved
##   ~(~a and ~b) == a or b
## which is different from a and b !
```

```
## Another way of looking at this: if a, b, c, ... are terms you add to the
## solver, the solver will attempt to find a satisfying assignment to
##  a and b and c and ...
## but if even one of a, b, c ... is false, then the whole formula is always
## false, e.g.
##   F and b and c and ... == F

## If x = ~a, y = ~b, z = ~c are the "theorems" you want to prove, then maybe
##       ~x and ~y and ~z
##    ==  a and  b and  c
##    ==  F and  T and  T
##    ==  F
## so the solver will return unsat even though "theorems" y, z are not true!

## Recall De Morgan's law : ~(x or  y) == ~x and ~y
##                          ~(x and y) == ~x or  ~y

## If we want to prove x and y and z, then we need to prove
##   ~(x and y and z) == ~x or ~y or ~z
## is unsat. This just says
##   "There does not exist an assignment in the theory for
##    which x is false or y is false or z is false"
```

Below are the Z3 arrays examples also from lecture.

```
from z3 import *

# Initialize arrays with types (sorts)

#                          Type of ...,     Type of ...
# Boolean Array            Sort of indices, Sort of values
boolArr = Array("boolArr", IntSort(),       BoolSort())
# boolArr = [True, False , True]

# Integers
arr = Array("arr", IntSort(), IntSort())
# arr = [1, 2 , 3]

# Return a new array that places 4 at index 1 in array arr
Store(arr, 1, 4) # arr [1] = 4
# If we access arr now, it will be the same as before the previous Store
# We need to explictly assign arr the new array
arr = Store(arr, 1, 4)

# Now arr has been updated with 4 at index 1
print(arr[1] == 4)
print(simplify(arr[1]))
```

```
## Axioms for Theory of Arrays

## Axiom 1: For all arr, i, v, we have
##      (Store(arr, i, v)) [i] == v

## Axiom 2: For all arr, i, j, v, we have
##      Store(arr, i, v)[j] == arr[j]   OR  i == j

## Explicit example
a = 1
b = 2
s = []
s.append(max(a,b))
print(s)

## Return a Z3 array tha places max(a,b) at dest in arr
def putMax(a, b , dest, array) :
    return Store(array, dest, max(a,b))

## Place max(a,b) into arr
arr = putMax(a,b, 0, arr)
print(arr)
## According to Axiom 1, this should be b
print(simplify(arr[0])) # arr[0] == b
## According to Axiom 2, this should be arr[1] (cannot simplify further)
print(simplify(arr[1])) # arr[1] == arr[1]

## Z3 arrays are infinite
bigindex = 99999999999999999999999999999999999999999999999999999999999999999
print(simplify(arr[bigindex])) # nothing stored yet
arr = Store(arr, bigindex, 17)
print(simplify(arr[bigindex])) # arr[bigindex] == 17

## Prove that arr[0] == b
solver = Solver()
solver.add(putMax(a,b,0,arr)[0] != b)
print(solver.check()) # unsat

## Counterexamples with Z3 and Arrays
tmpSolver = Solver()
tmpArr = Array("tmpArr", IntSort(), IntSort())
## Trying to prove arr[1] == b should return a counterexample
tmpSolver.add(putMax(a,b,0,tmpArr)[1] != b)
print(tmpSolver.check()) # sat
print(tmpSolver.model()) # K(Int, 3)
## K(Int, 3) is the constant array of 3s
```

```
## [3, 3, 3, .....]


## Let's prove some more general properties about arrays
## Prove that after
##    arr2[i] = max(a,b)
##    arr2[j] = min(a,b)
## then
##    arr2[i] <= arr[j]
## for any integers a, b, i, j
a = Int("a")
b = Int("b")
# dest = Int("dest")
maxIndex = Int("max") # i
minIndex = Int("min") # j

## Return max(x,y). If x == y, then this returns y
def z3Max (x, y) : return If(x >  y,  x, y)

## Return max(x,y). If x == y, then this returns y
def z3Min (x, y) : return If(x >= y, y, x)

arr2 = Array("arr2", IntSort(), IntSort())
solver2 = Solver()

arr2 = Store(arr2, maxIndex, z3Max(a,b))
arr2 = Store(arr2, minIndex, z3Min(a,b))

# print(simplify(Not(arr2[minIndex] <= arr2[maxIndex])))

solver2.add(Not(arr2[minIndex] <= arr2[maxIndex]))
print(solver2.check())
if solver2.check() == sat : print(solver2.model())
```

3 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.

3.1 Homework 0

Here are some web pointers to information that can help you with this homework. Regarding information about **argc** and **argv**, see:

```
https://port70.net/~nsz/c/c99/n1256.html#5.1.2.2.1
```

And, for **envp** (environment) information, see:

```
https://port70.net/~nsz/c/c99/n1256.html#5
https://port70.net/~nsz/c/c99/n1256.html#5.2
```

```
Homework Assignment 0
       CS 340d
 Unique Number: 52470
     Spring, 2021

Given:  January 19, 2021
 Due:  January 26, 2021
```

Simple C Program, Small Challenges

This homework concerns writing simple C programs. For this homework, you are to construct programs in response to the questions below.

This assignment is a warm-up exercise. We will be using C for various programming assignments. For those of you who have not used C, this assignment will give you a chance to familiarize yourself with the language.

This assignment has two parts.

Part A: Write a program that prints all of the arguments given to the "main" procedure of a C program. The procedure "main" is the first user-visible code that is run when the compiled result of C program is executed. When your program exits, the status code should return -1 if there was an error, otherwise it should return the value of the "main" formal parameter "argc".

Output for the environment variables should be handled as follows: Print the length of the value of each environment variable in a five character field, followed by the environment variable name and its value. For example, for the environment variable "TERM", you might see:

```
14 TERM xterm-256color
```

Part B: What does the following code do?

```
int has_what_property( unsigned long int x ) {
    return ((x - 0x101010101010101) & (~x) & 0x8080808080808080) != 0;
    }
```

Write an alternative specification, as a C-language definition that makes it clear what this code does. How could you confirm that the function has_what_property() meets your specification?

3.2 Homework 1

Homework Assignment 1
CS 340d
Unique Number: 52470
Spring, 2021

Given: January 21, 2021
Due: February 2, 2021

This homework assignment concerns comparing two C-languages programs that copy from file "input.txt" to file "output.txt".

Part A: Write a C-language program that copies the contents of input file "input.txt" and puts such contents into output file "output.txt". This will serve as your specification. Likely, you will wish to use C-library routines "getc" and "putc" which are defined in "stdio.h" and "stdlib.h".

Part B: We ask that you consider a C-language program that we claim copies the contents of input file "input.txt" and puts such contents into output file – but this program, shown below, does not refer to "stdio.h" and "stdlib.h".

This homework requires that you answer a number of questions about a file-copy program. This other file-copy program does not include the following two lines:

```
#include <stdio.h>
#include <stdlib.h>
```

Thus, the "standard" I/O libraries are not used. But, this code does use of the following three include files so we can use direct system-call-level procedures for I/O. We have provided an example file-copy program, and it uses several include files – which you will need to perform this homework.

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
```

As mentioned in Part A, we ask that you first write a file copy program using typical C-language library calls. This is your specification program. Your second program will be the result of modifying, if necessary, the copy program given below.

Part C: How can you compare the operation of these two programs to confirm that the program below works properly? Are there scenarios where the code below functions differently than your specification copy code? How can you convince a potential user of the copy code (below) performs exactly the same function as the code you created to satisfy Part A?

Part D: You must write a one- or two-page description (in ASCII only) of what had to be done to accomplish your task. For instance, how does the code below deal with the loss of the C-library functions? How did you confirm your program's correct operation?

You may find it useful to look at the include files: You can use the C-compiler itself to see the contents of the file the compiler will actually compile by:

```
gcc -E <c-program-file>.c
```

It is important that you explain why your code is correct. How can you establish the correctness of your solution? To start with, what is your specification?

For your writeup, we are not interested in formatting; thus, your writeup will actually be in the form of a C-language comment.

Note – everything in the file that you submit for this homework should be in ASCII. How can you check this? Is there a utility that can be used to confirm the "ASCII-ness" of your submission? Explain how you can guarantee that your submission is in ASCII only?

```
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>

int cp(const char *from, const char *to)
{

// Original "cp" example taken from Stack Overflow:
// https://stackoverflow.com/questions/2180079/how-can-i-copy-a-file-on-unix-using-c

// Code then modified to eliminate the use of goto.  Was this done correctly?

// Note the lack of references to library include files  <stdio.h> and <stdlib.h> .

  @strong{Question 1}  What does the C-language "const" keyword mean?
                      // Why is it used?

    int     fd_to;
    int     fd_from;
    char    buf[4096];

    @strong{Q1(a):}  Why this size?  How should this array be positioned?

    ssize_t nread;
    int     saved_errno;

    // Open input file

    fd_from = open(from, O_RDONLY);

    // If file open error, exit here
    if (fd_from < 0)
      return -11;

    // Open output file

    fd_to = open(to, O_WRONLY | O_CREAT | O_EXCL, 0666);
    if (fd_to < 0)
      {
```

```
    // The exit code idiom just below is repeated three times...
    saved_errno = errno;

    close(fd_from);
    if (fd_to >= 0)
      close(fd_to);

    // This way, the output file open return code is made
    // available, and not (likely successful) error code for
    // closing the input file.

    @strong{Question 2}  // Has something been lost?  Is there a reason we
                         // would want to see the input file return code?
    errno = saved_errno;
    return -12;
  }

while (nread = read(fd_from, buf, sizeof buf), nread > 0)
// While content is successful read, do:
  {
    char *out_ptr = buf;
    ssize_t nwritten;

    do {
      nwritten = write(fd_to, out_ptr, nread);

      if (nwritten >= 0)
        {
          nread -= nwritten;
          out_ptr += nwritten;
        }

      // Check if error return code means the system call was
      // interrupted by a signal.  If so, continue; otherwise,
      // quit.

      @strong{Question 3}  // This is a bit subtle -- can you explain the
                           // reason for this?
      else if (errno != EINTR)
        {
          // Exit...
          saved_errno = errno;

          // Again, closing the input file, but reporting a file
          // write problem.
          close(fd_from);
          if (fd_to >= 0)
```

```
              close(fd_to);

            errno = saved_errno;
            return -13;
          }
      } while (nread > 0);
    }

  if (nread <= 0)
    // When no characters are read or there is a read call error,
    // then close the output file and then close the input file.
    {
      if (close(fd_to) < 0)
      {
          fd_to = -1;

          // Exit...
          saved_errno = errno;

          // Close the input file.
          close(fd_from);
          if (fd_to >= 0)
            close(fd_to);

          // Depart with -14 status code.
          errno = saved_errno;
          return -14;
      }
      close(fd_from);

      /* Success! */
      return 0;
    }
  else {
    // Executing this code seems like a compiler error.

    @strong{Question 4}  // Can this program ever return a -15 status?
    return -15;
    }
  }


// To reduce the complexity of file name parsing, etc., I have
// ``hard-coded'' the input and output file names.  For your Homework
// #1 solution, you are welcome to do the same -- here, the input file
// is named ``input.txt'' and the output file is named ``output.txt''.
// This prevents a whole collection of potential problems with
```

```
// improper file names, buffer overruns, etc.

int main( int argc, char *argv[], char *env[] ) {

  char *input_file  = "input.txt";
  char *output_file = "output.txt";

  int cp_status = 0;

  cp_status = cp( input_file, output_file );

  return( cp_status );
}
```

3.3 Homework 2

Homework Assignment 2
CS 340d
Unique Number: 52470
Spring, 2021

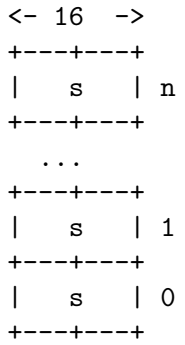
Given: January 28, 2021
Due: February 9, 2021

This homework concerns C-language types and predicates. Below are some diagrams. You must define a corresponding C-language type declaration for each diagram. Later there are some problems about type specifications. Remember to respect the organization of the fields within the structures – lower addresses are lower down on the page.

For problems 1, 2, 3, 4, and 5, write down a C type specification for the data structure shown or described. Abbreviations used:

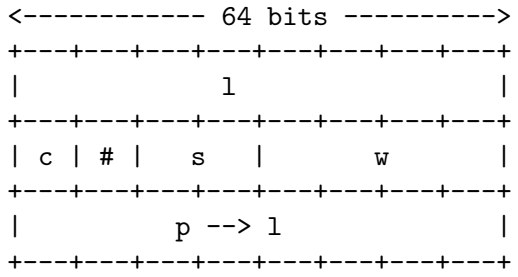
c - char s - short w - int l - long p - pointer # - no specification

1.



Assume n has been defined using #define macro.

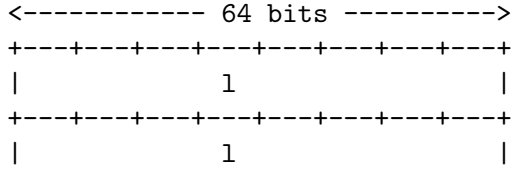
2. Define a C-language struct and a union for the following diagram.

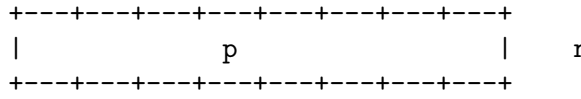


3. Define a type declaration for a pointer to an array of elements of the type in 2.

4. Define a type declaration for a linked-list of integers.

5.





Define a type, using "typedef" named "struct range", of the structure above, where "p" is a pointer to the type of the object above. When defining such a structure, name it "range" with field names "nx", "l", and "r" – and finally, with (type) name "range_t".

6. Given:

```
int a[] = { 3, 5, 8, 4, 2, 6, 7 };
```

what is:

```
a[*a + *(a + 4)]  ==  ?
```

7. What is the difference between type definitions a and b (just below):

```
a.  int (*months)[12]
b.  int *months[12]
c.  Draw a diagram that exhibits the two types above.
```

8. Diagram (meaning, draw a diagram like in problem 2), the following types:

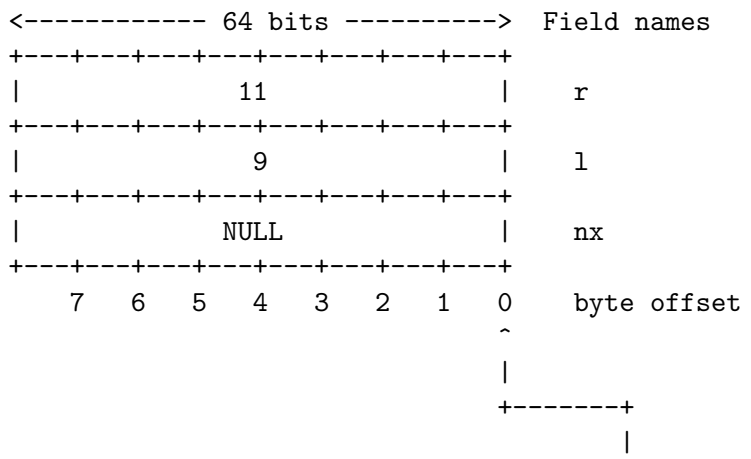
```
 a:   char    *x
 b:   char    *x()
 c:   char   (*x())
 d:   char   (*x())[]
 e:   char (*(*x())[])
 f:   char (*(*x())[])()
```

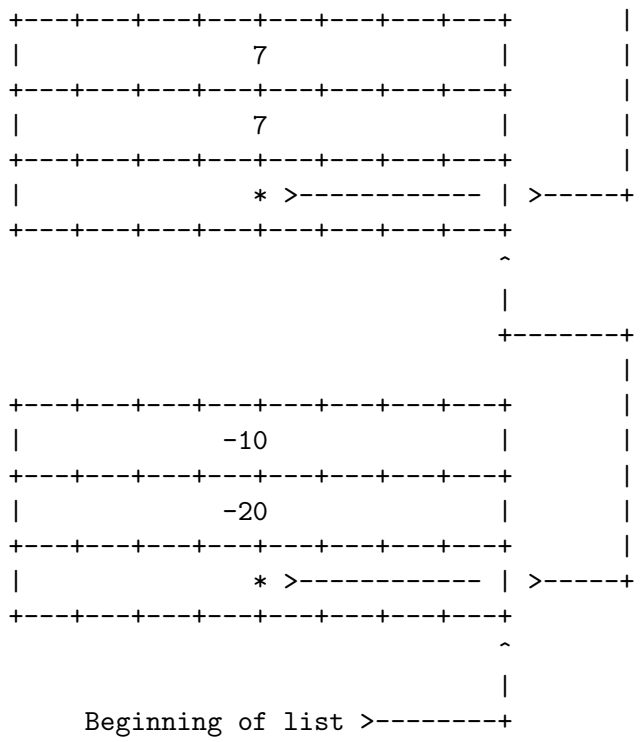
9. Is a freebie...

10. Define a subroutine that takes two "long" integer arguments, and uses "malloc" to allocate space for one structured object of the kind in problem 5 (above). The "p" field should be set to NULL, and this subroutine should return the address of the structure allocated.

The problems above are each worth one point. The next problem is worth ten points.

11. Consider a representation of a set that involves ranges of values on the number line. Our number line admits ranges from -2^63 to 2^63-1, inclusive. Here is an example representation for a set that includes a range from -20 to -10 (inclusive), 7, 9 to 11 (inclusive).





Given a list of pairs of numbers (each appearing on a separate line), create an ordered, compressed representation for the specified set. For example, the set above might have been specified (in a file) as:

```
-14 -10
7 7
11 10
9 10
-20 -12
```

Write a program that reads such a file and stores these contents as a linked list where each "struct range" is created through a call of "malloc". After reading the input file, your program should "clean" the data structure so that it is ordered (sorted) and so there are no range overlaps (between input pairs). For instance if the input file contained

```
2 3
4 5
7 10
9 14
```

then the final result should be

```
2 5
7 14
```

You are asked to write a C-language predicate that recognizes when the input is well-formed. Somewhere in your program, before you print the output, you should include an assert statement that calls your well-formed, compressed-element-recognizer predicate.

Finally, you need to print out the ranges, just like the example above, from smallest to largest.

We will run your program on inputs different than those in the example given just above. We will check to see that your program rejects improper inputs. In other words, you should have a predicate that recognizes valid input values. Identify this predicate in your code.

Before you start, the first thing to do is to think. Is the input well-formed? How would you determine this? Can you write a predicate that rejects ill-formed input files? What should you do when your input predicate recognizes malformed input? Overall, we are asking that you produce three predicates:

- a predicate that recognizes well-formed input,
- a predicate that recognizes well-formed data structure, and
- a predicate that recognizes an ordered version of data structure, which will be used with the predicate just above.

OK, once you believe you have a good input file, can you read the input into the data structure diagrammed above, and then print it out so that it exactly matched the input? Remember to define a predicate for the data structure above. Why would you do this? Answer: to help make sure that you read the input correctly. This should be available by way of the -c command-line flag option.

Finally, you need to compress overlapping ranges and order the cells of your data. Once done, you should print out the compressed ranges in order – from the most negative value to the most positive value – by printing two numbers per line for all (compressed) ranges. And, again, you should have a predicate that recognized an ordered version of the data structure above.

The program takes one command line argument – the file name – and can it should tolerate the "-c" argument to regurgitate its input as text.

The class TA will post corrections and turn-in instructions on Canvas.

3.4 Homework 3

Homework Assignment 3
CS 340d
Unique Number: 52470
Spring, 2021

Given: February 4, 2021
Due: March 2, 2021

This assignment concerns using the C-language standard library macro **assert** to write pre-conditions and post-conditions for various C-language standard library functions.

For each library function listed below, write the most complete pre-condition and post-condition you can. See the class lecture notes section titled **C Lanugage Assertions** for an example.

1. int **isspace**(int c)

This is very much like the example given in the class notes.

2. size_t **strlen**(char *cs)

This function takes a pointer to a string and returns the length. Does the length include the final (terminating) zero?

3. int **strncmp**(char *cs, char *ct, int n)

This requires recognizing strings. Note also, that the comparison may terminate prior to reaching the nth character.

4. char ***getenv**(const char *name)

What is the return type? If you attended class, this should be straightforward, but involve some work.

5. long **atol**(const char *s)

The **atol** has an Interesting pre-condition.

6. Explain why we chose to break a check for upper-case characters into the disjunction of three range checks (see just below).

```
int check_upper_char_2( char ch ) {
  // Upper-case character?
  return( 'A' <= ch && ch <= 'I' || 'J' <= ch && ch <= 'R'
                                 || 'S' <= ch && ch <= 'Z' );
}
```

For extra credit, consider writing a predicate that recognizes a well-formed input first argument for the function **printf**, where the **printf** call has exactly two arguments.

For the string (first) argument given **printf**, you should generate additional C-language code (using the format string) that checks that the two arguments (**arg1** and **arg2**) are well formed. The amount of extra credit is only bounded by your enthusiasm, accuracy,and quality of your answer.

Thus, you are given a call that looks like:

```
  char format_str = "<format string for arg1 and arg2>"
```

```
printf( format_str, arg1, arg2 );
```

Your result should be something like:

```
int analyze_format_str_arg1_arg2( char *fmtstr, void arg1, void arg2 );

// Prior to the printf, we perform an entry check...
assert( analyze_format_str_arg1_arg2( format_str, arg1, arg2 ) );

// And, then, we call the ‘‘printf’’ procedure...
printf( format_str, arg1, arg2 );
```

3.5 Homework 4

```
Homework Assignment 4
CS 340d
Unique Number: 52470
Spring, 2021

Given:  February 11, 2021
Due:  March 9, 2021
```

This homework concerns implementing the first three items of Lab #1: command-line processing, reading initial y86 memory contents from a file, and writing memory contents to a file. See See Section 4.2 [Lab 1 y86 Simulator], page 107, for details.

3.6 Homework 5

```
Homework Assignment 5
       CS 340d
 Unique Number: 52470
     Spring, 2021

Given:  March 4, 2021
 Due:  March 30, 2021
```

This homework involves specifying the pre- and post-conditions for two C-library memory copy routines, **memcpy** and **memmove**. In addition, you are asked to implement your own versions, **my_memcpy** and **my_memmove**, that perform exactly the same functions as the corresponding C-library procedures.

memcpy(char* s, char *ct, size_t n) is a memory copy routine that expects the source (pointed to by **ct**) and the target (pointed to by **s**) ranges to be completely disjoint from each other. Write entrance and exit predicates for **memcpy**; that is, write an **assert** statement that would precede a call to **memcpy** and an **assert** statement that would execute just after the **memcpy** procedure returns.

memmove(char* s, char *ct, size_t n) is a memory copy routine that allows the source range (from **ct** to **ct+n**) and destination range (from **s** to **s+n**) to overlap. This kind of copy is subtle because of how one copies when the target range overlaps the source range. Write entrance and exit predicates for **memmove**; that is, write an **assert** statement that would precede a call to **memmove** and an **assert** statement that would execute just after the **memmove** procedure returns.

This is the hard part of this assignment. Implement your own versions of **memcpy** and **memmove**; your routines should be called **my_memcpy** and **my_memmove**, and they should take exactly the same arguments as their corresponding C-library procedures.

For each loop in your **my_memcpy** and **my_memmove** routines, define a measure that decreases with each iteration – your loop(s) should iterate **n** times, copying one byte with each iteration.

The structure of your memory copy routines should be as follows.

```
void *my_memcpy( char *s, char *ct, size_t n ) {

  assert(
          // Everything you need to know upon entrance to this
          // routine.  This will be the same as the entrance
          // predicate you wrote for using memcpy.
          );

  // Your code, including your measure and loop termination assertion.

  assert(
          // Everything you need to know upon exit from this
          // routine.  This will be the same as the exit
```

```
          // predicate you wrote for using memcpy.
          );
}
```

and

```
void *my_memmove( char *s, char *ct, size_t n ) {

  assert(
          // Everything you need to know upon entrance to this
          // routine.  This will be the same as the entrance
          // predicate you wrote for using memmove.
          );

  // Your code, including your measure and loop termination assertion.
  // Hint:  For this procedure, you may need to have more than one loop;
  //         thus, you need a proper measure for each loop you define.

  assert(
          // Everything you need to know upon exit from this
          // routine.  This will be the same as the exit
          // predicate you wrote for using memmove.
          );
}
```

Note, this assignment, done well, will require very careful work and a good bit of time. In fact, this assignment will count as two homework grades; extra time is being given to complete this assignment.

The entrance and exit assertion approach mentioned above is what Microsoft did to improve the quality and reliability of the Windows operating system. I heard through the *grapevine* that Windows operating-system subroutines were five to ten pages in length, and often the input and exit assertions far exceeded the amount of code that was to be executed.

We will test the correctness of your implementation extensively, so it is important that your code is correct. In addition, in an ASCII-only comment should be included in the same file where your implementations for **my_memcpy** and **my_memmove** reside. In your comment, you are to explain why you believe your **assert** statements will never fire, why your code will terminate, and why your code is correct. Your comment should be provided in the form of a single C-language comment; it should be at least 90 lines but not more than 120 lines in length.

3.7 Homework 6

```
Homework Assignment 6
       CS 340d
 Unique Number: 52470
     Spring, 2021

Given:  March 25, 2021
  Due:  April 6, 2021
```

There are two parts to this assignment.

Part A:

Write a C-language, insertion-sort program that operates in-situ. Your program must include pre- and post-conditions. Please write your insertion sort with two routines:

```
insert -- inserts one item into a growing array
isort -- sorts an array of numbers by repeatedly inserting
         one item into an already sorted (sub-) list.
```

Also, demonstrate measures that decreases as these procedures execute. Below is a C-language template for your code.

```
// A Very Simple C Program

#include "stdio.h"


int array[100000];

void insert( int a[], unsigned int l, unsigned int r, int v ) {
  // Preconditions


  // Insert first item of sub-array into (ordered) remainder of array;
  // show decreasing measure


  // Postconditions
}

void isort( int a[], unsigned int l, unsigned int r ) {
  // Preconditions


  // A loop that repeatedly calls insert; show decreasing measure


  // Postconditions
```

```
}

int main()
{
  isort( array, 30, 999 );  // Example call
  printf( "Sub-Array Sorted!\n" );
}
```

Please include instructions for compiling (e.g., "gcc isort.c") and running a test on your solution to the problem above. Here is an example of the input format expected:

./a.out -a [2, 1, 4, 3]

By running your insertion sort program ("./a.out"), it should return

[1, 2, 3, 4]

Part B:

This portion of the assignment is to ensure you are familiar with the Python 3 enviornment. Why? We will use Python as our interface to the Z3 SMT system. Z3 can decide many conjectures that are helpful to confirm when programming. To get you ready to use Z3, we provide some problems.

1. Get a Python environment set up. What this looks like is up to you, but remember that we will be running all your programs on UTCS machines by simply invoking "python3 foo.py" where "foo.py" is your program. To get started, write a program that writes "I understand that all programs I write will be graded and tested on UTCS machines." to the terminal.
2. The l1 (read "ell one") norm of a vector x := [x1 x2 ... xn] computes the sum of the absolute values of the vector, i.e.,

   l1([x1 x2 ... xn]) = |x1| + |x2| + ... + |xn|

Write a function that computes the l1 norm of a vector and answer the following questions based on your function:

  (a) What is the domain (inputs) of your function?

  (b) What is the range (outputs) of your function?

  (c) Is your function correct? How do you know?

3. Suppose now all xi are valued 0 or 1. We then call x a bitvector. Let H denote the l1 norm for bitvectors.

Write a function that takes an integer and computes H on its binary representation.

  (a) What is the domain (inputs) of your function?

  (b) What is the range (outputs) of your function?

  (c) Input a value outside of the domain of H to your function. What happens?

```
  (d) Is your function correct? How do you know?

4. If you haven't already, SSH into the UTCS department machines.
   Launch the Python 3 shell by typing "python3" and run the following
   commands:

  >>> from z3 import *
  >>> p = Bool('p')
  >>> q = Bool('q')
  >>> s = Solver()
  >>> s.add(Not(And(p,q)) != Or(Not(p), Not(q)))
  >>> s.check()

  (a) What is returned after the last command?

  (b) What does that mean?  Write a few words about what you think
      just happened.

5. Place all your programs (questions 1-4) in a single .py file and
   ensure that they execute in the order the questions are given.
   Answer the questions as comments in the .py file.  For question 2
   and 3, the program should prompt the user to input a list of
   numbers and a single number, respectively, e.g.,

>>> I understand that all programs I write will be graded and tested on UTCS machines.
>>> Provide a list of numbers:
      1 2 3
>>> 6
>>> Provide a single number:
      2
>>> 1
>>> <question 4 output here ... etc.>
```

3.8 Homework 7

```
Homework Assignment 7
       CS 340d
 Unique Number: 52470
     Spring, 2021

Given:  April 6, 2021
 Due:  April 20, 2021
```

This homework involves specifying the pre- and post-conditions for some routines in the Class Assembler. Our y86 class assembler is available from the Piazza website that concerns our CS340d class.

For information about our Class Assembler, please read the C-style comments at the beginning of the Class Assembler file.

This assignment concerns creating a file that contains various pre- and post-conditions. Calls to these pre- and post-conditions have been included in the Class Assembler. In the Class Assembler, there is a C-language-style "#include" statement where you are expected to specialize a filename so it contains your UTEID (embedded as a part of the filename).

The file you are being asked to update is reproduced below. A link for the class assembler can be found on the top-level web page for our class. Please download it and the associated file named **student_uteid.h**. For this homework, your task is to finish the work that was started in this file.

You will turn in a new file named **student_uteid.h**, where the five characters **uteid** have been replaced with your personal **uteid**.

3.9 Homework 8

```
                        Homework Assignment 8
                               CS 340d
                         Unique Number: 52470
                             Spring, 2021

                        Given: April 15, 2021
                         Due:  April 27, 2021


CS 340d
Assignment 08

1. Recall from the last assignment, we defined the l1 norm for bitvectors.
Suppose we further restrict our domain to 32-bit integers, i.e.,

   x := [x1 ... x32]

Remember from your previous courses that there are lots cool hacks that you can
do with bitvectors. Given that x is 32-bit, the following funny looking code
computes H(x):

  def H(x) :
    x = (x & 0x55555555) + ((x >> 1)  & 0x55555555)
    x = (x & 0x33333333) + ((x >> 2)  & 0x33333333)
    x = (x & 0x0f0f0f0f) + ((x >> 4)  & 0x0f0f0f0f)
    x = (x & 0x00ff00ff) + ((x >> 8)  & 0x00ff00ff)
    x = (x & 0x0000ffff) + ((x >> 16) & 0x0000ffff)
    return x

  (a) Explain what the above code does by adding comments to each line. How does it
      compute the l1 norm for 32-bit strings?

  (b) Write pre- and post-conditions for this function in the form of assert statements.



2. Hypothetically, let's say you have two friends, Jan and Amanda, with whom you
are completing this assignment.  Jan and Amanda are very smart; before they even
saw the code in question 1, they came up with the following code that they
claim computes H for 32-bit integers:

  ## Jan's function
  def JanH(x) :
    x = x - ((x >> 1) & 0x55555555)
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333)
```

```
    x = (x + (x >> 4)) & 0x0f0f0f0f
    x = x + (x >>  8)
    x = x + (x >> 16)
    x = x + (x >> 32)
    x = x & 0x7f
    return x

  ## Amanda's function
  def AmandaH(x) :
    x = x - ((x >> 1) & 0x55555555)
    x = (x & 0x33333333) + ((x >> 2) & 0x33333333)
    x = (((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) & 0xffffffff) >> 22
    return x
```

Unfortunately, you suspect one of these functions is buggy, but you don't know whose. Moreover, since Jan and Amanda are so smart, you don't want to accuse their code of being wrong unless you have proof of a bug. You need to find a counterexample (i.e., some input on which the function is wrong).

(a) One way to find a counterexample would be to compare both JanH(x) and AmandaH(x) to H(x) for all 32-bit integers x. Try this. Does anything unexpected happen?

(b) Use Z3 to help you figure out whose function is buggy and give a counterexample as proof.

(c) Use Z3 to help you debug the buggy function and fix it.

(Hint: find out on which inputs the buggy function and H(x) differ, then compare the outputs in binary.)

(d) Use Z3 to show that the once buggy function is now correct after the fix.

Let's not think about the metaphysical consequences of Jan and Amanda reading question 2.

3. Below is some code to get you started. Write a Python comment to explain your use of Z3, e.g., how you used it in Question 2, why what you did in Question 2 proved what was desired, whether you found Z3 useful, if you would use it in other scenarios, etc.

```
from z3 import *
```

```
def H(x) :
  x = (x & 0x55555555) + ((x >> 1)  & 0x55555555)
  x = (x & 0x33333333) + ((x >> 2)  & 0x33333333)
  x = (x & 0x0f0f0f0f) + ((x >> 4)  & 0x0f0f0f0f)
  x = (x & 0x00ff00ff) + ((x >> 8)  & 0x00ff00ff)
  x = (x & 0x0000ffff) + ((x >> 16) & 0x0000ffff)
  return x

def JanH(x) :
  x = x - ((x >> 1) & 0x55555555)
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333)
  x = (x + (x >> 4)) & 0x0f0f0f0f
  x = x + (x >>  8)
  x = x + (x >> 16)
  x = x + (x >> 32)
  x = x & 0x7f
  return x

def AmandaH(x) :
  x = x - ((x >> 1) & 0x55555555)
  x = (x & 0x33333333) + ((x >> 2) & 0x33333333)
  x = (((x + (x >> 4) & 0xf0f0f0f) * 0x1010101) & 0xffffffff) >> 22
  return x
```

3.10 Homework 9

```
                        Homework Assignment 9
                              CS 340d
                         Unique Number: 52470
                             Spring, 2021

                        Given:  April 22, 2021
                         Due:    May 6, 2021
```

```
cs340d
Assignment 09

1. Use Z3 to prove the following theorem:

((p -> q) -> (r -> s)) & (s ->t) -> ((p -> q) -> (r -> t))

See propositional formula 3.77.2 (p.g. 55 in the notes)


2. Use Z3's theory of arrays to write a memcpy() program with the same
specification as in Assignment 5. Then write a Z3 "post-condition" that returns
unsat if the copy was successful and correct. Explain why your "memcopy"
function terminates. You do not need to write a pre-condition or use assert
statements, but you may if you wish. Below is some code to get started.
from z3 import *

## Store n data into memory at address
## Written recursively since "Store" returns a new Array
def store(data, address, n, mem) :
  if (n == 0) :
      return mem
  else :
      return store(data[1:], address + 1, n - 1, Store(mem, address, data[0]))

## Copy n "bytes" of data from address source to address dest in the memory model mem
def copy(source, dest, n, mem) :
  if (n == 0) :
      return mem
  else :
      return store( source + 1, dest + 1, n - 1, Store(mem, dest, mem[source]))

## Writes a formula to solver that is unsat iff the copy is correct (think post-condition)
def copyOK(source, dest, n, mem, solver) :
    return 0
```

```
## Initialise a Z3 array which will be our memory model
mem = Array("mem", IntSort(), IntSort())
## Initialise a solver
s = Solver()

## Some test "bytes"
data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

## store 10 "bytes" of data into mem starting at
mem = store(data, 0, 10, mem)

## copy 10 "bytes" of data from address 0 to address 20
# mem = copy(0, 20, 10, mem)

## Write a Z3 formula to solver s that returns unsat iff the copy was
## successful and correct.
# copyOK(0, 20, 10, mem, s)

## Check if the copy was successful
ch = s.check()
print(ch)

## Print out a counterexample if the copy was unsuccessful
if ch == sat : print(s.model())
```

4 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.

4.1 Lab 0 mywc

Laboratory 0
CS 340d
Unique Number: 52470
Spring, 2021

Given: January 28, 2021
Due: February 11, 2021

This laboratory concerns duplicating much of the functionality of the Unix "wc" command. In addition, this laboratory requires adding *additional* functionality.

General Comment

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 requirments, 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 programs, but that would require us to teach mathematics for most of semester. 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 sometimes refer to programs that can be specified formally.

Unfortunately, we will not write formal (mathematical) specifications for your programming assignments. Instead, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level commands. The running of these commands and their outputs will serve as executable simulators. Note, in this class, we will not concern ourselves with Windows behavior, but if you get this assignment to work with Windows also, that could be worth extra credit.

Laboratory Requirements

This laboratory involves duplicating the functionality of the "wc" command, including the command-line arguments "-c", "-l", "-w", but not the "-m" option, as your solution needs to work only on byte-oriented input streams or files. Note, your program should also work on binary files. You should name your "wc" command "mywc".

Here, we include some typical text that might help. 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 "mywc" 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

Linux, FreeBSD, or MacOS commands. What is a discrepancy? Absolutely any input file or stream that caused your "mywc" version 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.

In addition, your "mywc" program, when given the "-C" (the uppercase "C") option, should eliminate the remainder single-line C-language comment strings. As an example, when using the "-C" option, the lines:

```
Some example text // to be deleted
but not from this line.
```

should be processed as if your "mywc" program received:

```
Some example text
but not from this line.
```

Note, the final space character on the first line remains – and needs to be included in the character count your "mywc" returns.

When the "-C" option is given, your character and word count should not include the elided characters and words. In file, single-line, C-language-style comments start with the string "//" (two "/" characters) and extend to the end of the line. However, the end-of-line (<LF>) characters should not be elided. So, really, your specification is:

```
sed 's://.*$::g'  |  wc <options>
```

For more "sed" details", see: http://www.grymoire.com/Unix/Sed.html and see: http://sed.sourceforge.net/sed1line.txt . There are many more documents describing the regular-expression language. Note, we may study regular expressions in more depth later in the semester. So this is a good warm up exercise for that.

Laboratory Documentation

Finally, you need to include in your solution program a 60-line to 90-line comment as a C-language comment that begins with a line containing only "/*" and ends with a line containing only "*/" that describes your "mywc" 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.

Grading

You laboratory will be graded as with the following weights:

70% - Functioning of your "mywc" 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 (grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

Turn-in

Prior to the due date, we will post submission instructions.

4.2 Lab 1 y86 Simulator

```
Laboratory 1
CS 340d
Unique Number: 52470
Spring, 2021

Given:  February 11, 2021
Due:  March 11, 2021
```

This laboratory concerns implementing a simulator for y86 binary programs. The "General Comment" section (just below) is the same as it was in Lab 0; we repeat it in hopes that our comments about writing accurate specifications is beginning to seep into your consciousness.

General Comment

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 requirments, 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 programs, but that would require us to teach mathematics for most of semester. 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 sometimes refer to programs that can be specified formally.

Unfortunately, we will not write formal (mathematical) specifications for your programming assignments. Instead, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level commands. The running of these commands and their outputs will serve as executable simulators. Note, in this class, we will not concern ourselves with Windows behavior, but if you get this assignment to work with Windows also, that could be worth extra credit.

Laboratory Requirements

This laboratory involves implementing a binary-level simulator (emulator, interpreter) for the y86 computer model that was discussed in your CS429 class. We will simplify this assignment by truncating all memory addresses to 24-bits when accessing a byte array that we will use for the y86 memory; thus, an array of 2^24 bytes will be used for the simulated memory.

There are four parts to this assignment. Two of them are straight forward; they concern reading a file with the initial contents of memory and writing the contents of memory to a file. The third part concerns reading the command-line arguments for the initial RIP. These three parts are straightforward; you should do this immediately.

You can check whether your reader (memory fill) routine and your writer (memory dump) routine work correctly by creating test files, and seeing that your writer produces an equivalent output to your input.

The fourth part involves writing a y86 simulator so as to execute y86 binary programs.

The y86 class simulator, that each of you will write, involves four steps:

- Reading command-line arguments
- Initializing y86 simulator memory
- Running the y86 simulator
- Printing the contents of non-zero y86 memory locations

The first thing to do is to implement the command-line argument parser; see the source outline for additional details.

The second thing to do is to implement the memory-dump routine (writer).

The third thing to do is to implement the memory-fill routine (reader).

After you are sure that the three items above work perfectly, you can turn your attention to your y86 binary-program simulator.

We will be providing additional information about this part of the laboratory. We will review the code skeleton given below in class.

Laboratory Documentation

Finally, you need to include in your solution program file a 90-line to 120-line comment as a C-language comment that begins with a line containing only "/*" and ends with a line containing only "*/" that describes your "y86" 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.

Grading

You laboratory will be graded as with the following weights:

70% - The functioning of your "y86" implementation. 30% - The written description of your "y86" simulator.

Be careful with what you write. We will be grading the functioning of your program on multiple input files. And, we will carefully read your documentation, looking for problems (grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

Turn-in

Prior to the due date, we will post submission instructions.

```
************************************************************

Remember:  Look for bugs!!!

************************************************************

// y86.c                          For UT CS340d, by Warren A. Hunt, Jr.

// Version 0.4, March 4, 2021

#include "stdio.h"
```

```
#include "stdlib.h"
#include "ctype.h"
#include "assert.h"
#include "string.h"


// Some Constants

#define NUM_OF_REGS    (15)
#define NUM_OF_BYTES   (1 << 24)
#define MEM_ADDR_MASK  (NUM_OF_BYTES - 1)

#define MS_HALT        (1)
#define MS_ILLEGAL     (2)
#define MS_DECODE_REG_ERROR   (3)


// Here is the structure of the y86 interpreter state.

struct y86 {
  long int rgf[ NUM_OF_REGS ];  // The register file
  long int rip;                 // The instruction pointer, should this be unsigned?
  int zf;                       // The zero flag
  int sf;                       // The sign flag
  int of;                       // The overflow flag
  char mem[ NUM_OF_BYTES ];     // The memory, just the first 2^24 bytes, should mem be uns
  int ms;                       // The model state
};

struct y86 st;                  // Declare the y86 state

// Forward procedure references
char y86_mem_read( struct y86 *stp, long addr );
void y86_mem_write( struct y86 *stp, long addr, char byte );


void y86_clear( struct y86 *stp ) {
  // Clear the y86 state
  long i;

  // Clear register file
  for( i = 0; i < NUM_OF_REGS ; i++ )
    stp->rgf[ i ] = 0;

  // Clear RIP
  stp->rip = 0;
```

```
  // Clear Flags
  stp->zf  = 0;
  stp->sf  = 0;
  stp->of  = 0;

  // Clear Memory
  for( i = 0; i < NUM_OF_BYTES ; i++ )
    stp->mem[ i ] = 0;

  // Clear the y86 model state
  stp->ms = 0;
}

void y86_print_registers( struct y86 *stp, FILE *output_stream ) {
  // Print register file
  int i;
  for( i = 0; i < NUM_OF_REGS ; i++ )
    fprintf( output_stream, "R%d: %ld\n", i, stp->rgf[ i ] );

  // Print RIP
  fprintf( output_stream, "RIP: %ld\n", stp->rip );

  // Print Flags
  fprintf( output_stream, "Zero: %d, Sign: %d, Overflow: %d\n",
          stp->zf, stp->sf, stp->of );

  // Print y86 model state
  fprintf( output_stream, "Model state: %d\n", stp->ms );
}

void y86_print_memory( struct y86 *stp, FILE *output_stream,
                       long int start, long int end ) {
  // Print Memory
  int i;
  for( i = start; i < end ; i++ )
    fprintf( output_stream, "Mem[%d]: %d\n", i, stp->mem[ i ] );
  }


#define MAX_INPUT_LINE_LEN (30)

void y86_file_read( struct y86 *stp, FILE *input_stream ) {
  // Read an input file, which is a list of input pairs formated as follows:
  //     Each <address_i> is a natural number, 0..2^24-1, inclusive
  //     Each <value_i> is a natural number, 0..255, inclusive

  // Memory address will be monotonically increasing, with no duplicates.
```

```
// 012345...   Column numbers; this line is not part of the input file.
// (
//  (<address_0> <value_0>)
//  (<address_1> <value_1>)
//  (<address_5> <value_5>)
//  (<address_9> <value_9>)
//  ...
//  (<address_n> <value_n>)
// )

// No input lines should contain an <address>-<value> pair where
// <value> is 0.  Actually, the format above doesn't requre line
// breaks, but having this input format is exactly the output format
// -- so it is easy to confirm that the file reading and printing
// routines work correctly.

// Question:  What if a pair with <address_7> is specified two or more times?
// Answer:  Present error, and stop.  Note, with our format, we won't be able
//          to tell if an address is set to zero more than once.

// Read the input file into the memory.

char buf[MAX_INPUT_LINE_LEN];  // Line buffer
char junk[MAX_INPUT_LINE_LEN];
char *ans;                     // Answer pointer
int nargs;                     // Number of arguments

unsigned long addr = 0;
unsigned int  byte = 0;

// The largest 64-bit address is:  18446744073709551616
// The largest  8-bit value   is:  255
//   thus, the longest input line is:

// 012345...   Column numbers; this line is not part of the input file.
//  (18446744073709551616 255)

// Read and check first line of input
ans = fgets( buf, MAX_INPUT_LINE_LEN, input_stream );

assert( ans != NULL && ans == buf );
// printf( "1: %s", ans );   // To be deleted...
assert( ~strncmp( "(\n", ans, 3 ) );

// Read second line of input, remember the read includes the <lf>
ans = fgets( buf, MAX_INPUT_LINE_LEN, input_stream );
```

```
    while( ans != NULL && ans == buf && strnlen( ans, 5 ) > 3 ) {
    assert( strnlen( buf,  MAX_INPUT_LINE_LEN ) <= MAX_INPUT_LINE_LEN );

    // Read address-value pair...

    // Need code here!!!

    // Read
    ans = fgets( buf, MAX_INPUT_LINE_LEN, input_stream );
  }
  assert( ~strncmp( ")\n", ans, 3 ) );
}

void y86_file_write( struct y86 *stp, FILE *output_stream ) {
  // Output the final memory in the same format as the input fomat,
  // but only non-zero memory values should be written
  long i;

  fprintf( output_stream, "(\n" );
  for( i = 0; i < NUM_OF_BYTES; i++ ) {
    if ( stp->mem[ i ] != 0 )
      fprintf( output_stream, " (%lu %u)\n", i, (unsigned char) stp->mem[ i ] );
  }
  fprintf( output_stream, ")\n" );

}


// START of y86 simulator definition

char y86_mem_read( struct y86 *stp, long int addr ) {
  // Read y86 memory
  unsigned long u_addr = addr;
  return( stp->mem[ ( u_addr & MEM_ADDR_MASK ) ] );
}

void y86_mem_write( struct y86 *stp, long int addr, char byte ) {
  // Write y86 memory
  unsigned long u_addr = addr;
  stp->mem[ ( u_addr & MEM_ADDR_MASK ) ] = byte;
}


long int y86_mem_read_64( struct y86 *stp, long int addr ) {
  // Read y86 memory
  unsigned long u_addr = addr;
  char byte0 = y86_mem_read( stp, u_addr     );
```

```
  char byte1 = y86_mem_read( stp, u_addr + 1 );
  char byte2 = y86_mem_read( stp, u_addr + 2 );
  char byte3 = y86_mem_read( stp, u_addr + 3 );
  char byte4 = y86_mem_read( stp, u_addr + 4 );
  char byte5 = y86_mem_read( stp, u_addr + 5 );
  char byte6 = y86_mem_read( stp, u_addr + 6 );
  char byte7 = y86_mem_read( stp, u_addr + 7 );

  long int quadword = ( byte0       ||
                        byte1 <<  8 ||
                        byte2 << 16 ||
                        byte3 << 24 ||
                        (long int) byte4 << 32 ||   // Strange warning
                        (long int) byte5 << 40 ||
                        (long int) byte6 << 48 ||
                        (long int) byte7 << 56 );
  return( quadword );
}



void y86_step( struct y86 *stp ) {
  // Get first byte of instruction to simulate
  unsigned long rip = (unsigned long) stp->rip;

  unsigned char byte_at_pc = (unsigned char) y86_mem_read( stp, rip );
  unsigned char reg_byte = 0;
  unsigned char reg_byte_a = 0;
  unsigned char reg_byte_b = 0;

  // Much to complete below!!!

  switch( byte_at_pc ) {
    // Halt
  case 0: stp->ms = MS_HALT; break;

    // NOP
  case 16: stp->rip = stp->rip + 1; break;

    // RRMovL
  case 32:
    stp->rip = stp->rip + 1;
    reg_byte = y86_mem_read( stp, stp->rip );

    reg_byte_a = (reg_byte >> 4) & 0xF;
    reg_byte_b =  reg_byte       & 0xF;
```

```
    // Test for some stuff...
    if( reg_byte_a == 15 || reg_byte_b == 15 ) {
      stp->ms = MS_DECODE_REG_ERROR;
      break;}

    stp->rip = stp->rip + 1;
    // Do the work!
    stp->rgf[ reg_byte_b ] = stp->rgf[ reg_byte_a ];
    break;

    // IRMovL
  case 48:
    break;

    //RMMovL
  case 64:
    break;

    // MRMovL
  case 80:
    break;

    // Arithmetic operations
  case 96:  // Add
    break;

  case 97:  // Sub
    break;

  case 112:  // Jump
    stp->rip = 0;   // The new PC, whatever it should be...
    break;

  case 128:  // Conditional move ...

    break;
    // ...

  case 144:  // Subroutine Call

    break;


  default:
    stp->ms = MS_ILLEGAL;
    printf( "Very strange instruction.\n" );
    break;
```

```
  }
}

void y86( struct y86 *stp, size_t cnt ) {
  // Check if time remaining
  if ( cnt == 0 ) return;
  // Check if error set
  if ( stp->ms ) return;
  // Execute one instruction
  y86_step( stp );

  y86( stp, cnt - 1 );
}


int main( int argc, char *argv[], char *envp[] ) {

  // assert(  main_args_p( argc, argv, envp ) );

  size_t n = 0;  // Should be set by the ``-count'' command-line argument

  // Think about what other command-line arguments one might need.
  // For instance, is an initial RIP needed?

  // The size of the y86 state
  // printf("Overall size of y86 state is:  %ld.\n", sizeof( st ) );

  long int variable;
  printf( "Size of LONG INT is:  %ld.\n", sizeof( variable ) );


  y86_clear( &st );              // Clear the y86 state.

  // Initialize the memory.

  // st.mem[ 13 ] = 3;
  // st.mem[ 18 ] = 8;
  // y86_mem_write( &st, (unsigned long) 4611686018427387905, 1 );
  // y86_mem_write( &st, (unsigned long) 4611686018427387906, 2 );

  y86_file_read( &st, stdin );    // Fill memory

  //   y86, <n>
  y86( &st, 2 );                  // Run y86 interpreter on <n> instructions...

  // char *str = "a\n";
  // printf( "Length of ``%s'' is: %ld.\n", str, strnlen( str, 10 ) );
```

```
  // printf( "EOF:  %d.\n", EOF );

  y86_print_registers( &st, stdout );
  fprintf( stdout, "\n" );
  y86_print_memory( &st, stdout, 0, 32 );

  // Print all of the memory
  // y86_file_write( &st, stdout );  // Print memory

  return( st.ms );
}
```

4.3 Lab 2 y86 Debugger

```
Laboratory 2
CS 340d
Unique Number: 52470
Spring, 2021

Given:  March 22, 2021
Due:  April 20, 2021
```

This laboratory concerns implementing a debugger addition to your y86 simulator. The "General Comment" section (just below) is the same as it was in Lab 0; we repeat it in hopes that our comments about writing accurate specifications is beginning to seep into your consciousness.

General Comment

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 requirments, 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 programs, but that would require us to teach mathematics for most of semester. 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 sometimes refer to programs that can be specified formally.

Unfortunately, we will not write formal (mathematical) specifications for your programming assignments. Instead, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level commands. The running of these commands and their outputs will serve as executable simulators. Note, in this class, we will not concern ourselves with Windows behavior, but if you get this assignment to work with Windows also, that could be worth extra credit.

Laboratory Requirements

This laboratory involves implementing a binary-level debugger extension to your simulator (emulator, interpreter) for the y86 computer model that was discussed in your CS429 class.

There are two parts to this assignment: (Part 1) to write a test program that exercises every y86 instruction provided by your y86 simulator, and (Part 2) to implement various y86, binary-level debugging commands. The class y86 simulator (version 0.5) provides a template for every instruction that must be implemented.

Your test program should follow the same format as before (with the address-value pairs in decimal), but to make your program readable, you should also include comments for each instruction. For example,

```
(
```

```
 (<address_0> <value_0>)  ; e.g., irmovl 5, rbx
 (<address_1> <value_1>)
 ...
 (<address_n> <value_n>)
)
```

Of course, your simulator should ignore these comments. You should also give a brief description of what your program does. For a more complete description of how instructions are encoded in memory, see Computer Systems: A Programmer's Perspective (CSAPP), Third Edition.

Lab 1 assignment was hard, and this was reflected in the submissions. Some Lab 1 submissions didn't even provide implementations for all of the instructions! Obviously, we need a better result if you are going to use your simulators to debug y86 binary-level programs! We don't want to be debugging our simulators when we are trying to find subtle bugs in y86 binary-level programs.

Part 1 is due two weeks from the assignment date. Part 2 is due four weeks from the assignment date. We will test your Part 1 submission much more thoroughly than we did for Lab 1. You will resubmit Part 1 with Part 2 in four weeks from the assignment date.

Part 2 involves adding commands to your simulator so it can be controlled from the command line. To make this possible, the first update you need to make to your simulator is to implement the "-avf" address-value-file argument. This command-line flag requires an associated filename "<avf_filename>.avf" argument, such as:

```
--avf <av_filename>.avf
```

Such an "avf" (address-value-file) argument contains the initial memory contents for your y86-binary simulator. Thus, your simulator will not accept its initial memory content from standard-in, but from the filename specified.

Your simulator will now become interactive. It will accept the following commands:

```
a <a>   ; Set address to natural number: <a>

b <n>   ; Print <n> address-byte values (<addr> <byte>) starting at address <a>
q <n>   ; Print <n> address-quad values (<addr> <qword>) starting at address <a>
i       ; Print all registers, flags, the :RIP values

e <n>   ; Execute <n> y86 instructions
```

Note, the (separate) **a** command sets the address for the next (or repeated) use of the **b** and **q** commands. Using these commands, you will be able to exercise your simulator with various test programs, and you will use these commands to debug binary programs.

Adding the following commands will might it easier to use your simulator, but they are not required.

```
a <a>   ; Set address to natural number: <a>  [Same as above]
p       ; Set y86 simulator value :RIP to <a>
r <n>   ; Set y86 simulator register <a> to <n> (where <a> < 15)
B <v>   ; Write natural number memory byte <v> at <a>
Q <v>   ; Write natural number  memory quad-word <v> at <a>
```

To help you debug your simulator, we will provide the CS340d Class Assembler. The Class Assembler, provided by Alec Perry, is described in Homework #7. It will help you to do

Homework #7 as soon as possible as it involves the very Assembler that will generate code for your simulator. Here is some example input for the Class Assembler:

```
initialize:
     irmovq     1024 %rsp       # Initialize :RSP
     irmovq     1 %rdx          # Constant 1

operands:
     irmovq     20 %rax         # Number to add
     pushq      %rax            # Push on stack
     irmovq     13 %rax         # Second number to add
     pushq      %rax            # Push on stack

code-to-fetch-add-store:
     popq       %rcx            # Get second number
     popq       %rbx            # Get first number

# addl :rbx :rcx # Specification
loop:
     rrmovq     %rcx %rax       # Copy
     andq       %rax %rax       # Number zero?
equal-then-exit:
     je         store-answer    # If so, finished
add-1-sub-1:
     addq       %rdx %rbx       # Add 1 to first number
     subq       %rdx %rcx       # Subtract 1 from second number
     jmp        loop            # Repeat

store-answer: # Finished
     pushq      %rbx            # Push answer on stack
     halt
```

Below is a sample execution of your simulator using the commands above. First, the call:

```
  %  y86_sim --avf <y86_assembler>.avf
```

Once started, then your simulator should accept the commands above so you can run the code just loaded into your y86 simulator. Here is an example interactive session; the lines that start with "**>**" characters are output from your simulator.

```
  ./y86_sim --avf <y86_assembler>.avf
  i
> ((:RIP 0)
>  (:RAX 0 :RBX 0 :RCX 0 :RDX 0
>   :RDI 0 :RSI 0 :RBP 0 :RSP 0
>   :R08 0 :R09 0 :R10 0 :R11 0
>   :R12 0 :R13 0 :R14 0)
>  (:F-ZF 0 :F-SF 0 :F-OF 0)
>  (:MR-STATUS 0))
  a 0
> 0
```

```
  e 6
> 6
  i
> ((:RIP 44)
>  (:RAX 13 :RBX 0 :RCX 0 :RDX 1
>   :RDI 0 :RSI 0 :RBP 0 :RSP 1008
>   :R08 0 :R09 0 :R10 0 :R11 0
>   :R12 0 :R13 0 :R14 0
>  (:F-ZF 0 :F-SF 0 :F-OF 0)
>  (:MR-STATUS 0))
  a 1008
> 1008
  q 2
> ((:ADDRESS 1008 :QWORD-VALUE 13)
>  (:ADDRESS 1016 :QWORD-VALUE 20))
  ... And so on...
```

Laboratory Documentation

Finally, you need to include in your solution program file a 90-line to 120-line comment as a C-language comment that begins with a line containing only "/*" and ends with a line containing only "*/" that describes your "y86" command and describes how to use your simulator.

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.

When turning in your lab, make sure that your documentation addresses how to use your simulator.

Grading

You laboratory will be graded as two parts with the following weights:

```
Part 1:

  70% - The correct functioning of your "y86" binary-level simulator
        and a test program.

  30% - The written description of your "y86" binary-level simulator,
        and why your test program is thorough.

Part 2:

  70% - The functioning of your "y86" debugger commands

  30% - The written description of your "y86" debugger commands
```

Be careful with what you write. We will be grading the functioning of your program on multiple input files. And, we will carefully read your documentation, looking for problems (grammar, spelling, run-on sentences, tense agreement, etc.) – errors will lower your grade.

Turn-in

Prior to the due date, we will post submission instructions.

4.4 Lab 3 y86 Debugging

```
Laboratory 3
CS 340d
Unique Number: 52470
Spring, 2021

Given:  April 20, 2021
Due:  May 4, 2021
```

This laboratory concerns using your y86 simulator and your y86 debugger to debug some y86 binary code.

The "General Comment" section (just below) is the same as it was in Lab 0; we repeat it in hopes that our comments about writing accurate specifications is beginning to seep into your consciousness.

General Comment

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 requirments, 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 programs, but that would require us to teach mathematics for most of semester. 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 sometimes refer to programs that can be specified formally.

Unfortunately, we will not write formal (mathematical) specifications for your programming assignments. Instead, we will provide executable predicates (to recognize desired validity conditions) and simulators in the form of various combinations of Linux/MacOS/FreeBSD user-level commands. The running of these commands and their outputs will serve as executable simulators. Note, in this class, we will not concern ourselves with Windows behavior, but if you get this assignment to work with Windows also, that could be worth extra credit.

Laboratory Requirements

This laboratory requires you to reverse-engineer and debug y86 binary programs, and determine whether they work properly. These programs are given at the end of this Lab assignment.

For each program, you are asked to reverse-engineer the y86 “binary” code into y86 assembler. And, for each program, you are asked to describe what function the binary program provides and how one uses the binary program. If there is a bug in any of these programs, you should provide a fix. You should provide example usage for each program. Note, the first program starts at zero. The second program does not start at zero, but very close to its start, it initializes the stack pointer.

Laboratory Documentation

Finally, you need to describe what each program does, and what fixes, if any, were required to get it to work properly. For each program you reverse engineer, and fix as required, you should describe your observations in a 60-line comment as a C-language comment that begins with a line containing only "/*" and ends with a line containing only "*/" that describes your use of your commands and what each program does.

Your description of each program and what it does 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.

Grading

You laboratory will be graded as two parts with the following weights:

```
70% - The correct functioning of your y86 binary programs, with fixes,
      if required.  And, please provide a test program for each program.

30% - The written description of what each y86 binary program does, and
      your process of bug discovery and repair.
```

Be careful with what you write. We will be grading the functioning of your program on multiple input files. And, we will carefully read your documentation, looking for problems (grammar, spelling, run-on sentences, tense agreement, etc.) – remember, errors may lower your grade.

Turn-in

Prior to the due date, we will post submission instructions.

Binary Programs

Below are some binary programs to reverse engineer. Note, there is a bug in the second program; hopefully, you can find it and fix it.

It will probably be helpful to reverse-engineer this into assembler, and get the class assembler to produce this exact binary. Then, you will have source code that will be easier to understand.

Program 1:

```
(
 (0 99)
 (2 48)
 (3 242)
 (4 9)
 (12 99)
 (13 102)
 (14 48)
 (15 241)
 (16 1)
 (24 97)
 (25 22)
 (26 96)
```

```
 (27 32)
 (28 96)
 (29 98)
 (30 118)
 (31 26)
)
```

Program 2:

```
(
 (0 160)
 (1 95)
 (2 32)
 (3 69)
 (4 160)
 (5 63)
 (6 160)
 (7 111)
 (8 80)
 (9 83)
 (10 8)
 (18 99)
 (20 98)
 (21 51)
 (22 113)
 (23 96)
 (31 48)
 (32 240)
 (33 1)
 (41 32)
 (42 49)
 (43 97)
 (44 1)
 (45 115)
 (46 96)
 (54 160)
 (55 31)
 (56 128)
 (65 176)
 (66 31)
 (67 32)
 (68 6)
 (69 48)
 (70 241)
 (71 2)
 (79 97)
 (80 19)
 (81 160)
```

```
 (82 63)
 (83 128)
 (92 176)
 (93 31)
 (94 96)
 (95 96)
 (96 176)
 (97 111)
 (98 176)
 (99 63)
 (100 32)
 (101 84)
 (102 176)
 (103 95)
 (104 144)
 (112 48)
 (113 244)
 (115 32)
 (122 32)
 (123 69)
 (124 48)
 (125 240)
 (126 3)
 (134 160)
 (135 15)
 (136 128)
 (145 176)
 (146 63)
)
```

Doc Index

Z