St. Francis Xavier University
Department of Computer Science

CSCI 544: Computational Logic
Group Lecture and Report
Winter 2024

The group lecture and report are major assessments of this course. These assessments consist of three components: a lesson plan, the lecture itself, and an individual report. The group lecture is preferably completed in a group of four students, though *very limited* exceptions can be made if a group of four cannot be formed.

The lesson plan is worth a total of 10% of your final grade. The lecture component is worth a total of 15% of your final grade. The individual report is also worth a total of 15% of your final grade. You must complete all of these components in order to pass the course.

Do not leave these assessments until the last minute! Start as soon as possible!

Lesson Plan

Due March 6, 2024 in lecture

The lesson plan is a short document meant to establish your group and your group's chosen lecture topic.

The lesson plan should include, at a minimum, the following information:

- The names of your group members
- One or two paragraphs giving a high-level introduction to your group's lecture topic
- A description of what each group member plans to contribute to the lecture
- A broad outline of the structure of your group's lecture (e.g., X minutes for introduction, Y minutes for background, etc.)
- A list of preliminary references that your group will use to create your lecture materials

A title page is not required.

When planning what each group member can contribute to the lecture, consider splitting the lecture topic into independent sections or subtopics and assigning one section or subtopic to each group member.

Existing survey articles or books are great preliminary references, as they often include a large amount of background information presented at an accessible level. If your group needs pointers to such preliminary references, let me know and I can try to help.

All references must be cited in a consistent style (e.g., IEEE citation style). Citations to articles and books are preferred over citations to lecture notes, Wikipedia, or websites.

Your group's lesson plan should be a 1–2 page double-spaced document written in 12-point text. This is a strict page limit, but references do not count toward the limit. The lesson plan will be marked in terms of completeness, organization, and adherence to guidelines.

Each group only needs to submit one lesson plan for all group members.

Lecture

Due date varies (last ∼3 weeks of lectures)

The lecture is a ∼50 minute presentation on your group's chosen topic. Your group will not be presenting for the entire length of time; you should budget ∼5 minutes for setup at the beginning and ∼5 minutes for questions and teardown at the end. Therefore, your actual lecture time will be closer to ∼40 minutes, and each group member should spend an approximately equal amount of time speaking (i.e., in a group of four, each group member speaks for ∼10 minutes).

Group members should take turns presenting so that no one is unfairly burdened with having to speak for the entire duration. This is an opportunity for individuals to use their strengths; for example, one person may cover the introduction, one person may work through an example, one person may choose to lead a live coding session, etc.

Your group's lecture should be presented at a level similar to other lectures in this course: targeted at a graduate student audience that is familiar with computational logic. Your group can refer to past lecture notes as background material, but your group should ultimately develop your own notes or slides for use in the lecture itself.

Your group must submit your lecture materials (e.g., notes, slides, etc.) *at least one day before* the lecture is scheduled. Your lecture will be marked in terms of three broad categories: quality (e.g., focused on topic, targeted at an appropriate level), accuracy (e.g., correct summary of main results, material all related to topic), and organization (e.g., notes/slides are clearly written, lecture is well-paced).

All lecture material must be appropriately referenced, and citations must be included wherever necessary.

Lastly, remember that *quality is better than quantity*. A very well-produced lecture that takes ∼30 minutes is better to experience than a poorly-planned lecture that takes ∼40 minutes.

Report

Due April 10, 2024 in office hours

The report is an individual submission meant to complement the group lecture. In this report, you will write a brief survey of your lecture topic *in your own words* and give a short reflection on your group lecture experience.

On the first page of your report, you should include, at a minimum, the following information:

- The names of your group members and the topic of your lecture
- A list of the specific contributions you made in your group
- Approximately one paragraph discussing aspects of your lecture that you feel went well and aspects of your lecture that you feel could have been improved. Your opinion will not be shared with other group members.

In the rest of your report, you should include the following information:

- 3–4 pages summarizing your lecture topic, written in the style of a survey article or short lecture notes
- A complete list of references used in your report (not in your group lecture in general)

You must additionally include a title page that lists your name, report topic, and course details.

Note that your 3–4 page summary should be written *in your own words*; you must submit your own summary, and not one written collaboratively by all group members. If your group divided work by assigning specific sections or subtopics to individual members, this is a good opportunity to showcase your knowledge about the particular section or subtopic you focused on while designing your group lecture.

All references must be cited in a consistent style (e.g., IEEE citation style). Citations to articles and books are preferred over citations to lecture notes, Wikipedia, or websites.

Your report should be a 4–5 page double-spaced document written in 12-point text. This is a strict page limit, but references and the title page do not count toward the limit. The report will be marked primarily in terms of completeness, organization, and adherence to guidelines.

Since the report is due at the end of the term, the report deadline is firm. Late reports will not be accepted.

Suggested Topics

Below, I offer a selection of topics that your group can choose for your lecture. If your group wishes to study and present a topic related to this course that is not on this list, your group must check with me first to approve the proposed topic.

1. **Belief logic.**

   Belief logic is an extension of logic where, instead of studying what is or is not true or what follows from what, we study the properties of believing or willing statements. For example, in belief logic, one might express the notion that "you believe A is true" or "everyone believes that they ought to do A". One can then naturally introduce connectives to express notions like "If you believe A, then you do not believe B".

2. **Deontic and imperative logic.**

   Deontic logic is an extension of logic where the validity of a statement depends on permissions and obligations, such as "A is obligatory" or "it is permissible to do A". By contrast, imperative logic changes statements from being indicative to being imperative: instead of the statement "one is doing A", imperative logic would consider the statement "do A".

3. **Modal logic.**

   Modal logic is an extension of logic that introduces the notions of necessity and possibility to statements. For example, while we may have a statement that expresses "A is true", modal logic might consider statements like "A is possible" or "A is necessary". Expressing that something is possible is a weaker claim than expressing that thing is true, while expressing that something is necessary is stronger than expressing its truth, for you are asserting that the thing is not only true, but it must always be true.

4. **Temporal logic.**

   Temporal logic is an extension of logic that introduces rules for reasoning about statements in terms of time. For example, a traditional statement A always has the same meaning, but it may be true or false at varying points in time. Temporal logic introduces terms like "always", "sometimes", and "until" to formalize this idea that truth values may vary in this way.

5. **Linear-time temporal logic.**

   By combining modal logic and temporal logic, we can obtain a system called linear-time temporal logic (or LTL) that allows us to express formulas about conditions changing truth values over time. LTL is particularly useful in applications that involve paths or trees, and it has found use in the formal verification of computer programs and model checking.

6. **Hoare logic.**

   A very important aspect of software development is the ability to reason rigorously about the correctness of software. Hoare logic, introduced in 1969 by Tony Hoare, is a formal logical system that defines a set of rules to allow a programmer to formally prove that their software is correct. Hoare logic uses the notion of a "triple" $\{P\}C\{Q\}$ to encode a command C, preconditions P needed to execute the command, and postconditions Q arising as a result of the command's execution.

7. **Software verification, partial correctness, and total correctness.**

   Further extending the idea of reasoning rigorously about the correctness of software, one can define different notions of correctness. The notions of correctness relate closely to theoretical computer science and decidability: we say that an algorithm or a piece of software is "partially correct" when, *if* an output is produced, that output is correct; while an algorithm is "totally correct" when a correct output is *always* produced (that is, the algorithm always halts). Software verification relies on these two notions of correctness, though as a consequence of the halting problem, establishing total correctness is a process that can never be fully automated.

8. **The Coq proof assistant.**

   From the Coq website (`https://coq.inria.fr/a-short-introduction-to-coq`):

   "Coq is a proof assistant. It means that it is designed to develop mathematical proofs, and especially to write formal specifications, programs and proofs that programs comply to their specifications. An interesting additional feature of Coq is that it can automatically extract executable programs from specifications, as either Objective Caml or Haskell source code. Properties, programs and proofs are formalized in the same language called the Calculus of Inductive Constructions (CIC). Then, all logical judgments in Coq are typing judgments: the very heart of Coq is in fact a type-checking algorithm."

   Groups who choose this topic are expected to present a live demonstration of the software as part of their group lecture.

9. **The Isabelle/HOL proof assistant.**

   From the Isabelle website (`https://isabelle.in.tum.de/overview.html`):

   "Isabelle is a generic proof assistant. It allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. The main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer hardware or software and proving properties of computer languages and protocols. The most widespread instance of Isabelle nowadays is Isabelle/HOL, which provides a higher-order logic theorem proving environment that is ready to use for big applications."

   Groups who choose this topic are expected to present a live demonstration of the software as part of their group lecture.

10. **The Lean programming language/proof assistant.**

    From the Lean website (`https://lean-lang.org/about/`):

    "Lean is a functional programming language that makes it easy to write correct and maintainable code. You can also use Lean as an interactive theorem prover. Lean programming primarily involves defining types and functions. This allows your focus to remain on the problem domain and manipulating its data, rather than the details of programming. Lean has numerous features, including: type inference; first-class functions; powerful data types; pattern matching; type classes; extensible syntax; hygienic macros; dependent types; metaprogramming framework; multithreading; verification: you can prove properties of your functions using Lean itself."

    Groups who choose this topic are expected to present a live demonstration of the language/software as part of their group lecture.

11. **The Agda programming language.**

    From the Agda documentation (`https://agda.readthedocs.io`):

    "Agda is a dependently typed programming language. It is an extension of Martin-Löf's type theory. Because of strong typing and dependent types, Agda can be used as a proof assistant, allowing one to prove mathematical theorems (in a constructive setting) and to run such proofs as algorithms."

    Groups who choose this topic are expected to present a live demonstration of the language as part of their group lecture.

12. **The Prolog programming language.**

    From the Prolog documentation (`https://prolog.readthedocs.io`):

    "Prolog is a logic programming language used in fields like artificial intelligence. Facts and rules (i.e. Horn Clauses) are used to express the program logic in Prolog. Prolog computation involves querying a database of facts and rules. If a given query can be proven for a given database, Prolog outputs the answers for the query and a message like "Yes" or "true" to tell the user that the query was proven. If the query can't be proven, either a message like "false" or an error is outputted to the user."

    Groups who choose this topic are expected to present a live demonstration of the language as part of their group lecture.

13. **SAT solvers and the state of the art.**

    A SAT solver is a piece of software that is specially designed to solve instances of the Boolean satisfiability problem. Even though Boolean satisfiability is well-known to be NP-complete in general, very efficient and scalable algorithms have been developed to solve instances of the problem involving even thousands of variables. SAT solvers have had a dramatic impact on the fields of software verification, computer-assisted proving, and artificial intelligence, and some competitions have been held to advance the state-of-the-art in SAT solving.

14. **Binary decision diagrams.**

    A binary decision diagram (or BDD) is a special kind of binary tree that can be used to represent a Boolean function. In a BDD, leaves correspond to outputs of the Boolean function and internal vertices correspond to decisions made for individual variables of the Boolean function: for example, one branch of an internal vertex may correspond to a variable x being assigned "true", while the other branch corresponds to x being assigned "false". There are many variant types of BDDs, including but not limited to ordered BDDs and reduced ordered BDDs.