COMP 3030 – Automata Theory and Formal Languages

Course Description

Calendar entry

An introduction to automata theory, grammars, formal languages and their applications. Topics: finite automata, regular expressions and their properties; context-free grammars, pushdown automata and properties of context-free languages; Turing machines and their properties. Prerequisite: COMP 2080.

General Course Description

COMP 3030 introduces some of the central ideas of theoretical computer science. The course covers some formal models of computation, including the finite automaton, the pushdown automaton, and the Turing machine. For each model, students first learn how to design machines, and then learn to prove facts about which problems can and cannot be solved within the model. Some relationships between these formal models and practical applications such as lexical analysis, text editing, machine design, syntax analysis, parser generation are also covered.

Detailed Prerequisites

The primary prerequisite for this course is reasonable mathematical background. Students should feel comfortable with abstract mathematics and proofs. Specific topics that are useful include a knowledge of logic, sets, and functions, as well as basic data structures and algorithms.

Before entering this course, a student should be able to:

- Describe a set using set-builder notation, and be familiar with basic set operations (e.g., union, intersection, complement). Prove relationships between two given sets, e.g., subset, disjointedness, equality.
- Use and understand function notation, e.g., specifying the domain, co-domain, and range. Describe and prove whether or not a function is injective and/or surjective.
- Use and understand propositional and predicate logic.
- Prove mathematical propositions using various techniques: direct proof, indirect proof, proof by contradiction, proof by cases, proof by mathematical induction.
- Write pseudocode to precisely describe an algorithm. Prove that an algorithm correctly solves the intended task.

Course Goals

By the end of this course students will:

- Define and use several formal models of computation: finite automata, pushdown automata, Turing machines.
- For each formal model of computation studied, define machines to solve a given problem. Prove that the machine correctly solves the problem.
- For each formal model of computation studied, describe the set of problems that can and cannot be solved in the model.
- Use various techniques to prove impossibility, i.e., that a given problem cannot be solved by any machine within a particular formal model.
- Describe operations on formal languages, and prove closure facts about these operations. Use closure facts to prove whether or not a language belongs to a certain class.
- Understand the difference between determinism and non-determinism, and how it might affect computational power.
- Explain the historical context of computability in the study of mathematics, and how Alan Turing's work contributed to the development of the modern computers we use today.
- Relate formal models of computation to real-world systems and applications (e.g., state machines, compilers, pattern matching, software development tools).

Learning Outcomes

Terminology

Students should be able to:

1. Use notation and terminology related to strings (e.g., alphabet, prefix, suffix, substring) and string operations (e.g., concatenation, reverse).
2. Understand the concept of "formal language" and understand related operations on languages (e.g., union, intersection, concatenation, complement, product, Kleene star).
3. Use notation and terminology related to functions (e.g., input types, output types, decision problems, computability, encodings, reductions).

Finite Automata and Regular Expressions

Students should be able to:

1. Model a simple real-world system as an abstract state machine by defining the system inputs, the states of the system, and the transitions between states.
2. Formally define a finite automaton by providing mathematical definitions of the state set, the input alphabet, the transition function, the start state, and the accepting states. Draw an equivalent machine diagram representation.
3. Simulate the execution of a finite automaton on a given input string to determine the outcome.
4. Compare and contrast determinism vs. non-determinism in the context of finite automata (i.e., DFA's and NFA's).
5. Explain the relationship between finite automata and regular expressions.

6. Given a regular expression and a string, determine if the pattern matches the string.

Regular Languages

Students should be able to:

1. Given a regular language, design a corresponding finite automaton or regular expression.
2. Write a formal proof using state invariants that a specific finite automaton correctly decides a given regular language.
3. Given a finite automaton or regular expression, identify its corresponding regular language.
4. Use the Myhill-Nerode Theorem to determine the size of the smallest possible deterministic finite automaton (DFA) that decides a given regular language. (Time-permitting: Describe an algorithm that shrinks a given DFA to its smallest possible size.)
5. Use various techniques to prove that a given language is not regular, e.g., Pigeonhole Principle, Myhill-Nerode Theorem, the Pumping Lemma.
6. For a given operation on languages, prove whether or not the set of all regular languages is closed under the operation.
7. Use known closure facts to prove whether or not a given language is regular.

Context-Free Grammars

Students should be able to:

1. For a given grammar and string, write out a derivation of the string and draw the corresponding parse tree.
2. Understand the concept of ambiguity, and prove whether or not a given grammar is ambiguous.
3. Define normal forms (e.g., Chomsky, Greibach) and identify if a given grammar is in a particular normal form.
4. Describe an algorithm that converts any given grammar into Chomsky Normal Form.
5. Describe the CYK algorithm that decides whether a given grammar in Chomsky Normal Form generates a given string.

Pushdown Automata

Students should be able to:

1. Formally define a pushdown automaton by providing mathematical definitions of the state set, the input alphabet, the stack alphabet, the transition function, the start state, and the accepting states. Draw an equivalent machine diagram representation.
2. Simulate the execution of a pushdown automaton on a given input string to determine the outcome.
3. Compare and contrast determinism vs. non-determinism in the context of pushdown automata (i.e., DPDA's and PDA's)
4. Explain the relationship between pushdown automata and context-free grammars.

Context-free Languages

Students should be able to:

1. Given a context-free language, design a corresponding context-free grammar or pushdown automaton.
2. Given a context-free grammar or pushdown automaton, identify its corresponding context-free language.
3. Use the Pumping Lemma to prove that a given language is not context-free.
4. For a given operation on languages, prove whether or not the set of all context-free languages is closed under the operation.
5. Use known closure facts to prove whether or not a given language is context-free.

Turing Machines

Students should be able to:

1. Formally define a Turing machine by providing mathematical definitions of the state set, the input alphabet, the tape alphabet, the transition function, the start state, the accept state, and the reject state. Draw an equivalent machine diagram representation.
2. Simulate the execution of a Turing machine on a given input string to determine the outcome.
3. Draw a Turing machine diagram that decides a given language, or computes a given function.
4. Given a Turing machine diagram, identify the language it decides or the function it computes.
5. Discuss Turing machine variants and their equivalence to each other. Compare and contrast determinism vs. non-determinism in the context of Turing machines.
6. (Time-permitting) Define complexity classes related to running time or memory requirements of Turing machines.

Computability

Students should be able to:

1. Describe and use terminology relating to Turing-decidability and Turing-recognizability.
2. Understand universality and be able to describe/implement the encoding process of a specific object (e.g., a Turing machine). Describe how a Universal Turing Machine can simulate the execution of any other Turing machine.
3. Use Cantor's diagonal argument to prove the existence of non-computable functions and problems.
4. Give specific examples of undecidable problem (e.g., The Halting Problem), and use a reduction to prove that a given problem is undecidable.
5. Explain Rice's Theorem and its proof. For a given undecidable problem, argue whether or not Rice's Theorem can be applied.
6. Give specific examples of unrecognizable problems and use a reduction to prove that a given problem is unrecognizable.

7. Design an algorithm to prove that a given problem is decidable (or recognizable). Use the dovetailing technique to prove that a given problem is recognizable.
8. For a given operation on languages, prove whether or not the set of all decidable languages is closed under the operation, and prove whether or not the set of all recognizable languages is closed under the operation.
9. Use known closure facts to prove whether or not a given language is decidable (or recognizable).
10. (Time-permitting) Describe the Post Correspondence Problem and use it in reductions to prove the undecidability of various problems involving context-free languages (e.g., detecting ambiguity)
11. (Time-permitting) Describe the Busy Beaver Game, its related non-computable functions, and its application to solving open problems in mathematics.

Applications of Automata Theory and Formal Languages

Students should be able to:

1. Explain the use of finite automata and regular expressions in text searching tasks, and in the tokenizing step of the code compilation process.
2. Explain the use of pushdown automata and context-free grammars in the parsing step of the code compilation process.
3. Explain the Church-Turing Thesis (and its historical context), and how it relates the study of computability to software development (for example, the impossibility of fully-automated testing/debugging, or Turing-completeness).