CS420

Introduction to the Theory of Computation

Lecture 1: Introduction

Tiago Cogumbreiro
About the course

- **Location:** (Y01-1350) Room 1350, 1st floor, University Hall
  
  **SUBJECT TO CHANGE! We need a bigger room.**

- **Schedule:** Monday, Wednesday / 4:00pm to 5:15pm

**Instructor:**

- **Name:** Tiago (蒂亚戈) Cogumbreiro (he/him)
- **Email:** Tiago.Cogumbreiro@umb.edu
- **Office:** (M03-0201-16) Room 0201-16, 3rd floor, McCormack
- **Office hours:** TBD
Course webpage

- **URL:** cogumbreiro.github.io/teaching/cs420/f22/
- Holds the class schedule and the syllabus
Other resources

- gitlab.com: Homework assignment PDFs
- gradescope.com: Homework/mini-test submission site
- blackboard.com: Quiz submission site
- discord.com: Office hours, communication, Q&A

Make sure you have access to each of these sites!
Breadth versus depth

- Solve quizzes and mini-tests first, because they have a **hard deadline** of 24h.
- Solve homework assignments second, because they have a **soft deadline**.
  You can always resubmit any homework assignment.
- Prioritize (breadth) solving more assignments/exercises over (depth) solving single assignment/exercise flawlessly.

Don't forget to fill today's quiz in Blackboard!
Course requirements

Checklist

- Install Coq 8.15 (v2022.04.0): coq.inria.fr
- Can you access Gradescope?
- Can you access Blackboard?
- Can you access #cs420 and #cs420-news in Discord? If not ask in #cs420-lounge
- Can you access Gitlab? (The invites will be rolling out until this Friday)

Heads up

- Please, register using your UMB email address.
Course overview
Introduction to Theory of Computation

Formal Languages

- Understanding the limits of what computers and programs
  - Regular languages
  - Context-Free languages
  - Turing-recognizable languages
A birdseye view of CS420
What are the limits of programs?
Limits of computation

- Different classes of machines
- The limits of each of these classes
- What properties each class enjoys
Limits of computation

- Different classes of machines
- The limits of each of these classes
- What properties each class enjoys

Classes of machines

<table>
<thead>
<tr>
<th>Class of machine</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finite Automata</td>
<td>Parse regular expressions</td>
</tr>
<tr>
<td>Pushdown Automata</td>
<td>Parse structured data (programs)</td>
</tr>
<tr>
<td>Turing Machines</td>
<td>Any program</td>
</tr>
</tbody>
</table>
Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams
Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams

- **Regular expressions** (regex)
  String matching rules
Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams

- **Regular expressions** *(regex)*
  String matching rules

- **Grammars**
  Data specification; Parsing data
Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams

- **Regular expressions** *(regex)*
  String matching rules

- **Grammars**
  Data specification; Parsing data

- **Turing machines**
  Theory of computation
Techniques

- **State-machines**
  Structure concurrency/parallelism/User Interfaces; UML diagrams

- **Regular expressions** (regex)
  String matching rules

- **Grammars**
  Data specification; Parsing data

- **Turing machines**
  Theory of computation

- **Programs are proofs**
  Using a programming language to write formal proofs
Some applications of formal languages
Use Case 1: DFA/NFA

Using a DFA/NFA to structure hardware usage
Use Case 1: DFA/NFA

Using a DFA/NFA to structure hardware usage

- Arduino is an open-source hardware to design microcontrollers
- Programming can be difficult, because it is highly concurrent
- Finite-state-machines structures the logical states of the hardware
- **Input:** a string of hardware events
- String acceptance is not interesting in this domain

Example

The FSM represents the logical view of a micro-controller with a light switch
Use Case 1

Declare states

```c
#include "Fsm.h"

// Connect functions to a state
State state_light_on(on_light_on_enter, NULL, &on_light_on_exit);

// Connect functions to a state
State state_light_off(on_light_off_enter, NULL, &on_light_off_exit);

// Initial state
Fsm fsm(&state_light_off);
```

Source: [platformio.org/lib/show/664/arduino-fsm](platformio.org/lib/show/664/arduino-fsm)
Use Case 1

Declare transitions

```c
// standard arduino functions
void setup() {
    Serial.begin(9600);

    fsm.add_transition(&state_light_on, &state_light_off,
                       FLIP_LIGHT_SWITCH,
                       &on_trans_light_on_light_off);

    fsm.add_transition(&state_light_off, &state_light_on,
                       FLIP_LIGHT_SWITCH,
                       &on_trans_light_off_light_on);
}
```

Source: platformio.org/lib/show/664/arduino-fsm
Use Case 1

Code that runs on before/after states

// Transition callback functions
void on_light_on_enter() {
  Serial.println("Entering LIGHT_ON");
}

void on_light_on_exit() {
  Serial.println("Exiting LIGHT_ON");
}

void on_light_off_enter() {
  Serial.println("Entering LIGHT_OFF");
}

// ...

Source: platformio.org/lib/show/664/arduino-fsm
Use Case 2

Regular Expressions: Input validation
Use Case 2

Regular Expressions: Input validation

HTML includes regular expressions to perform client-side form validation.

```html
<input id="uname" name="uname" type="text"
    pattern="_([a-z][A-Z][0-9])+" minlength="4" maxlength="10">
```

- `_([a-zA-Z0-9]+` means any character between `a` and `z`, or between `A` and `Z`, or between `0` and `9`
- `R+` means repeat `R` one or more times
- In this case, the username must start with an underscore ``, and have one or more letters/numbers
- `minlength` and `maxlength` further restrict the string's length
Use Case 3

Regular Expressions: Text manipulation
Use Case 3

Regular Expressions: Text manipulation

Programming languages include regular expressions for fast and powerful text manipulation.

Example (JS)

```javascript
let txt1 = "Hello World!";
let txt2 = txt1.replace(/\[a-zA-Z]+/, "Bye"); // Replaces the first word by "Bye"
console.log(txt2); // Bye World!
```
Use Case 4

Parsing JSON
ANTLR is a **parser generator**.

- **Input:** a grammar; **Output:** a parser, and data-structures that represent the parse tree (known as a Concrete Syntax Tree)
- The HTML DOM is an example of an **Abstract** Syntax Tree

```antlr
json: value; // initial rule
obj: '{' pair (',' pair)* '}' | '{' '}'; // a sequence of comma-separated pairs
pair: STRING ':' value; // Example: "foo": 1
array: '[' value (',' value)* ']'; | '[' ']'; // a sequence of comma-separated values
value: STRING | NUMBER | obj | array | 'true' | 'false' | 'null'; // ...
```

Source: [raw.githubusercontent.com/antlr/grammars-v4/master/json/JSON.g4](https://raw.githubusercontent.com/antlr/grammars-v4/master/json/JSON.g4)
A grammar for JSON integers

```
NUMBER: '-'? INT ('.' [0-9]+)? EXP?

fragment INT: '0' | [1-9] [0-9]*; // fragment means do not generate code for this rule
fragment EXP: [Ee] [+-]? INT;     // fragment means do not generate code for this rule
```

Source: raw.githubusercontent.com/antlr/grammars-v4/master/json/JSON.g4
A grammar for JSON

```java
import org.antlr.v4.runtime.tree.ParseTreeListener;

// Generated from ../JSON.g4 by ANTLR 4.7.2
import org.antlr.v4.runtime.tree.ParseTreeListener;

/**
 * This interface defines a complete listener for a parse tree produced by
 * `{@link JSONParser}`.
 * */
public interface JSONListener extends ParseTreeListener {
    /**
     * Enter a parse tree produced by `{@link JSONParser#json}`.
     * @param ctx the parse tree
     */
    void enterJson(JSONParser.JsonContext ctx);

    /**
     * Exit a parse tree produced by `{@link JSONParser#json}`.
     * @param ctx the parse tree
     */
    void exitJson(JSONParser.JsonContext ctx);
}
- Study **algorithms** and **abstractions**
- Theoretical study of the **boundaries of computing**

---

CS420

Lecture 1

Tiago Cogumbreiro

UMass Boston
Course schedule

1. Learn the Coq programming language
2. Regular languages
   - Design state machines
   - Prove properties on regular languages
3. Context-free languages
   - Design pushdown automata
   - Prove properties on regular languages
4. Turing-machines
   - Prove properties on computable and non-computable languages
On studying effectively for this content

Suggestions

- **Read the chapter before the class:**
  This way we can direct the class to specific details of a chapter, rather than a more topical end-to-end description of the chapter.

- **Attempt to write the exercises before the class:**
  We can guide a class to cover certain details of a difficult exercise.

- **Use the office hours and our online forum:**
  Coq is a unusual programming language, so you will get stuck simply because you are not familiar with the IDE or a quirk of the language.
Module 1
Basics.v: Part 1

A primer on the programming language Coq

We will learn the core principles behind Coq
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?
Enumerated type

A data type where the user specifies the various distinct values that inhabit the type.

Examples?

- boolean
- 4 suits of cards
- byte
- int32
- int64
Declare an enumerated type

```
Inductive day : Type :=
   | monday : day
   | tuesday : day
   | wednesday : day
   | thursday : day
   | friday : day
   | saturday : day
   | sunday : day.
```

- **Inductive** defines an (enumerated) type by cases.
- The type is named `day` and declared as a `: Type` (Line 1).
- Enumerated types are delimited by the assignment operator (`:=`) and a dot (`.`).
- Type `day` consists of 7 cases, each of which is tagged with the type (`day`).
Printing to the standard output

Compute prints the result of an expression (terminated with dot):

```
Compute monday.
```

prints

```
= tuesday
: day
```
Interacting with the outside world

- Programming in Coq is different most popular programming paradigms
- Programming is an **interactive** development process
- The IDE is very helpful: workflow similar to using a debugger
- It’s a REPL on steroids!
- `Compute` evaluates an expression, similar to `printf`
Inspecting an enumerated type

```plaintext
match d with
| monday  ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday  ⇒ monday
| saturday ⇒ monday
| sunday  ⇒ monday
end
```
Inspecting an enumerated type

```c
match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end
```

- match performs **pattern matching** on variable d.
- Each pattern-match is called a **branch**; the branches are delimited by keywords `with` and `end`.
- Each **branch** is prefixed by a mid-bar (|) (⇒), a pattern (eg, `monday`), an arrow (⇒), and a return value.
Pattern matching example

```haskell
Compute match monday with
    | monday => tuesday
    | tuesday => wednesday
    | wednesday => thursday
    | thursday => friday
    | friday => monday
    | saturday => monday
    | sunday => monday
end.
```
Create a function

**Definition** next_weekday (d:day) : day :=

match d with
| monday ⇒ tuesday
| tuesday ⇒ wednesday
| wednesday ⇒ thursday
| thursday ⇒ friday
| friday ⇒ monday
| saturday ⇒ monday
| sunday ⇒ monday
end.
Create a function

**Definition** `next_weekday (d:day) : day :=`

```
match d with
| monday  => tuesday
| tuesday => wednesday
| wednesday => thursday
| thursday => friday
| friday  => monday
| saturday => monday
| sunday  => monday
end.
```

- **Definition** is used to declare a function.
- In this case `next_weekday` has one parameter `d` of type `day` and returns (:) a value of type `day`.
- Between the assignment operator (:=) and the dot (.), we have the body of the function.
Example 2

**Compute** \(\text{next\_weekday fri}day\).

yields (Message pane)

\[
\begin{align*}
= & \text{monday} \\
: & \text{day}
\end{align*}
\]

\text{next\_weekday fri}day \text{ is the same as mon}day (after evaluation)
Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
simpl. (* simplify left-hand side *)
reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.
Your first proof

Example test_next_weekday:
  next_weekday (next_weekday saturday) = tuesday.
Proof.
  simpl. (* simplify left-hand side *)
  reflexivity. (* use reflexivity since we have tuesday = tuesday *)
Qed.

- Example prefixes the name of the proposition we want to prove.
- The return type (:) is a (logical) proposition stating that two values are equal (after evaluation).
- The body of function test_next_weekday uses the ltac proof language.
- The dot (.) after the type puts us in proof mode. (Read as "defined below".)
- This is essentially a unit test.