

CS420

Introduction to the Theory of Computation

Lecture 1: Introduction

Tiago Cogumbreiro

About the course

- **Instructor:** Tiago (蒂亚戈) Cogumbreiro
- **Classes:** M01-0207, McCormack
4:00pm to 5:15pm, Monday, Wednesday
- **Office hours:** 1:00pm to 2:00pm, Monday, Wednesday, Friday

Course webpage

cogumbreiro.github.io/teaching/cs420/s20/

Syllabus

cogumbreiro.github.io/teaching/cs420/s20/syllabus.pdf

- Course divided into 4 modules
- 2 homework assignment + 1 mini-test per module (30mins)
- Final grade: 26% mini-tests + 70% homework + 4% participation
- **Homework grade:** average of **8 assignments** (possibly weighted)
- **Participation grade:** in-class quizzes, attendance classroom
- **Classroom attendance is required!**

Grade		Letter	
95 ≤	P		A
90 ≤	P	< 95	A-
85 ≤	P	< 90	B
75 ≤	P	< 85	B
70 ≤	P	< 75	B-
65 ≤	P	< 70	C+
55 ≤	P	< 65	C
50 ≤	P	< 55	C-
45 ≤	P	< 50	D+
35 ≤	P	< 45	D
30 ≤	P	< 35	D-
30 ≤	P		F

Course requirements

Checklist

- Install Coq 8.10: coq.inria.fr
- Sign in on Gradescope: www.gradescope.com/courses/81793
- Sign in on Piazza: piazza.com/class/k5ubsxch57r196
- Sign in on Estalee: www.estalee.com

Heads up

- Please, **register using your UMB email address.**
- **Homework 1** is due February 10 at 11:59pm.

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

<i>Class of machine</i>	<i>Applications</i>
Finite Automata	Parse regular expressions
Pushdown Automata	Parse structured data (programs)
Turing Machines	Any program

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

```
#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

Use Case 1

Declare transitions

```

// 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.

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

- `_[a-zA-Z0-9]+`
- `[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)

```
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

Grammar for 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

```

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

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

```

> ls *.java
JSONBaseListener.java JSONParser.java JSONVisitor.java
JSONBaseVisitor.java JSONLexer.java JSONListener.java
> cat JSONBaseListener.java
// 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);
  }

```

CS420

- Study **algorithms** and **abstractions**
- Theoretical study of the **boundaries of computing**

Course schedule

1. Learn Coq programming
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

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


Inspecting an enumerated type

```

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

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 (next_weekday friday).
```

yields (Message pane)

```
= monday  
: day
```

next_weekday friday is the same as monday (after evaluation)

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.

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.