COMP 302 Winter 2019 Lecture 1

Prakash Panangaden[1]

[1]School of Computer Science
McGill University

McGill University,Montréal, January 2019

Welcome to COMP 302

- My name: Prakash Panangaden

Welcome to COMP 302

- My name: Prakash Panangaden
- How should you address me?

Welcome to COMP 302

- My name: Prakash Panangaden
- How should you address me?
- Prof. Panangaden,

Welcome to COMP 302

- My name: Prakash Panangaden
- How should you address me?
- Prof. Panangaden,
- **not** Prof. Prakash!!

Welcome to COMP 302

- My name: Prakash Panangaden
- How should you address me?
- Prof. Panangaden,
- **not** Prof. Prakash!!
- Prakash (no title) is fine also

Welcome to COMP 302

- My name: Prakash Panangaden
- How should you address me?
- Prof. Panangaden,
- **not** Prof. Prakash!!
- Prakash (no title) is fine also
- as is Sir.

Course Title

Official Title

Programming Languages and Paradigms

TAs

1. Ariella Smofsky (head TA)
2. Aliya Hameer (will only interact with other TAs)
3. Kelvin Tagoe
4. Akshal Aniche
5. Nathaniel Bos
6. TBA

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.
2. Office hours: Mondays, Wednesdays 4:00 to 5:30
3. Office location: Room 105**N** McConnell

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.
2. Office hours: Mondays, Wednesdays 4:00 to 5:30
3. Office location: Room 105**N** McConnell
4. I will set up a MyCourses page with
   - links to the course web site

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.
2. Office hours: Mondays, Wednesdays 4:00 to 5:30
3. Office location: Room 105**N** McConnell
4. I will set up a MyCourses page with
   - links to the course web site
   - discussion boards

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.
2. Office hours: Mondays, Wednesdays 4:00 to 5:30
3. Office location: Room 105**N** McConnell
4. I will set up a MyCourses page with
   - links to the course web site
   - discussion boards
   - lecture recordings

Course Administration

1. `cs.mcgill.ca/~prakash/Courses/302/comp302.html`
   Lecture notes, assignments and solutions will be posted there.
2. Office hours: Mondays, Wednesdays 4:00 to 5:30
3. Office location: Room 105**N** McConnell
4. I will set up a MyCourses page with
   - links to the course web site
   - discussion boards
   - lecture recordings
5. There is a Facebook group.

Grading

- 8 assignments : 24% Submitted through an automated system using LearnOCaml.

Grading

- 8 assignments : 24% Submitted through an automated system using LearnOCaml.
- 3 quizzes : 6% using the myCourses system.

Grading

- 8 assignments : 24% Submitted through an automated system using LearnOCaml.
- 3 quizzes : 6% using the myCourses system.
- 1 in-class midterm: 10% of your grade,

Grading

- 8 assignments : 24% Submitted through an automated system using LearnOCaml.
- 3 quizzes : 6% using the myCourses system.
- 1 in-class midterm: 10% of your grade,
- Final exam: 60% of your total grade.
- Cheat sheets for exams: no other notes, no books, no calculators, phones, laptops, smart watches, Google glasses, mirrors or magic owls.

Paradigms

Official definition

a distinct concept or **thought pattern**

1. Functional programming: higher-order, polymorphically typed (OCaml)
2. Imperative programming (OCaml)
3. Object-oriented programming: inheritance and subtyping (Java)

What languages will we use?

- Answer 1: OCaml, Java
- Answer 2: not important!

What languages will we use?

- Answer 1: OCaml, Java
- Answer 2: not important!
- Anyone who describes this course as “Programming in OCaml” does not get it!

Topics

1. Recursion

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics
5. Higher-order functions

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics
5. Higher-order functions
6. Updatable data: references

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics
5. Higher-order functions
6. Updatable data: references
7. Environments and binding

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics
5. Higher-order functions
6. Updatable data: references
7. Environments and binding
8. Closures and objects

Topics

1. Recursion
2. how to think about it, not how it is implemented with stacks!
3. Inductively defined types and structures
4. Operational semantics
5. Higher-order functions
6. Updatable data: references
7. Environments and binding
8. Closures and objects
9. Some other topics

Topics II

1. Types, typing rules

Topics II

1. Types, typing rules
2. Type inference, polymorphism

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers
4. Object oriented paradigm

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers
4. Object oriented paradigm
5. Subtyping and inheritance

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers
4. Object oriented paradigm
5. Subtyping and inheritance
6. they are **NOT** the same thing!

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers
4. Object oriented paradigm
5. Subtyping and inheritance
6. they are **NOT** the same thing!
7. Stream programming, if time permits.

Topics II

1. Types, typing rules
2. Type inference, polymorphism
3. Interpreters, parsers and compilers
4. Object oriented paradigm
5. Subtyping and inheritance
6. they are **NOT** the same thing!
7. Stream programming, if time permits.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.
- In mathematical discussions we need to use precise mathematical notation.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.
- In mathematical discussions we need to use precise mathematical notation.
- The so-called “popular” books are more confusing than the real thing.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.
- In mathematical discussions we need to use precise mathematical notation.
- The so-called “popular” books are more confusing than the real thing.
- Computers are **completely unforgiving**!!

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.
- In mathematical discussions we need to use precise mathematical notation.
- The so-called “popular” books are more confusing than the real thing.
- Computers are **completely unforgiving**!!
- In order to train our minds we need to learn to speak carefully.

The role of language

- We all speak natural language(s) with varying degrees of precision and accuracy.
- But sloppy use of language causes confusion in mathematics.
- In mathematical discussions we need to use precise mathematical notation.
- The so-called “popular” books are more confusing than the real thing.
- Computers are **completely unforgiving**!!
- In order to train our minds we need to learn to speak carefully.
- Every sentence has to be constructed with care.

English sloppiness that drives me crazy

- Writing “can not” instead of “cannot”: causes a real mistake in meaning.

English sloppiness that drives me crazy

- Writing “can not” instead of “cannot”: causes a real mistake in meaning.
- Mistaking “that” and “which”; sloppy but we usually understand.

English sloppiness that drives me crazy

- Writing “can not” instead of “cannot”: causes a real mistake in meaning.
- Mistaking “that” and “which”; sloppy but we usually understand.
- “I would like eggs and bacon or sausages.” Ambiguity

English sloppiness that drives me crazy

- Writing “can not” instead of “cannot”: causes a real mistake in meaning.
- Mistaking “that” and “which”; sloppy but we usually understand.
- “I would like eggs and bacon or sausages.” Ambiguity
- “Dr. Lex Luthor is a former alumni of Gotham State.”

English sloppiness that drives me crazy

- Writing “can not” instead of “cannot”: causes a real mistake in meaning.
- Mistaking “that” and “which”; sloppy but we usually understand.
- “I would like eggs and bacon or sausages.” Ambiguity
- “Dr. Lex Luthor is a former alumni of Gotham State.”
- Clearly does not know what “alumnus” means nor what is the singular form. I saw this in a newspaper article.

What is the study of language?

- We are not studying a dozen languages to pad out your CV.

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?
- Commonly written in manuals,

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?
- Commonly written in manuals,
- but they are generally vague and ambiguous.

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?
- Commonly written in manuals,
- but they are generally vague and ambiguous.
- We will show you (a bit of) *formal* semantics

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?
- Commonly written in manuals,
- but they are generally vague and ambiguous.
- We will show you (a bit of) *formal* semantics
- and typing rules.

What is the study of language?

- We are not studying a dozen languages to pad out your CV.
- Linguistics: syntax and semantics.
- Syntax: what is a correctly formed sentence/program?
- Semantics: what does a sentence mean? (natural)
- Semantics: what does a program do when you run it?
- Commonly written in manuals,
- but they are generally vague and ambiguous.
- We will show you (a bit of) *formal* semantics
- and typing rules.

The role of abstraction

- Our only technique for handling complexity.

The role of abstraction

- Our only technique for handling complexity.
- Conceptualization with*out* reference to specific instances.

The role of abstraction

- Our only technique for handling complexity.
- Conceptualization with*out* reference to specific instances.
- Isolating the fundamental, essential issues without irrelevant details.

The role of abstraction

- Our only technique for handling complexity.
- Conceptualization with*out* reference to specific instances.
- Isolating the fundamental, essential issues without irrelevant details.
- When designing a data structure: where it will be used is not important.

The role of abstraction

- Our only technique for handling complexity.
- Conceptualization with*out* reference to specific instances.
- Isolating the fundamental, essential issues without irrelevant details.
- When designing a data structure: where it will be used is not important.
- When using a data structure: details of the implementation are not important.

The role of abstraction

- Our only technique for handling complexity.
- Conceptualization with*out* reference to specific instances.
- Isolating the fundamental, essential issues without irrelevant details.
- When designing a data structure: where it will be used is not important.
- When using a data structure: details of the implementation are not important.
- Thinking about all the details **is not a virtue**.

The abstraction principle

"Every significant piece of functionality should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts." — Benjamin C. Pierce

Why software engineers need math

"Software engineering is all about abstraction. Every single concept, construct and method is entirely abstract. Of course, it does not feel that way to most software engineers. But that's my point. The main benefit that they got from the mathematics they learned in academia was the experience of rigourous reasoning with purely abstract objects and structures." — Keith Devlin.

The basic pieces

1. Values

The basic pieces

1. Values
2. Names

The basic pieces

1. Values
2. Names
3. Variables (Locations)

The basic pieces

1. Values
2. Names
3. Variables (Locations)
4. Expressions

The basic pieces

1. Values
2. Names
3. Variables (Locations)
4. Expressions
5. Commands

Higher-level pieces

1. Combination mechanisms:
    1. control-flow constructs
    2. combinators

Higher-level pieces

1. Combination mechanisms:
   1. control-flow constructs
   2. combinators
2. Parametrized expressions = functions

Higher-level pieces

1. Combination mechanisms:
   1. control-flow constructs
   2. combinators
2. Parametrized expressions $=$ functions
3. Parametrized commands $=$ procedures (methods)

Higher-level pieces

1. Combination mechanisms:
   1. control-flow constructs
   2. combinators
2. Parametrized expressions $=$ functions
3. Parametrized commands $=$ procedures (methods)
4. Modules: independent compilation.

Types

1. Classify values

Types

1. Classify values
2. and expressions

Types

1. Classify values
2. and expressions
3. and functions and procedures

Types

1. Classify values
2. and expressions
3. and functions and procedures
4. in order to *restrict* what can be expressed.

Types

1. Classify values
2. and expressions
3. and functions and procedures
4. in order to *restrict* what can be expressed.
5. Give up expressive power for

Types

1. Classify values
2. and expressions
3. and functions and procedures
4. in order to *restrict* what can be expressed.
5. Give up expressive power for
6. guarantees of good behaviour.

What to understand

1. Names: binding and scoping

What to understand

1. Names: binding and scoping
2. Evaluation rules: expressions $\rightarrow$ values

What to understand

1. Names: binding and scoping
2. Evaluation rules: expressions $\rightarrow$ values
3. Typing rules,

What to understand

1. Names: binding and scoping
2. Evaluation rules: expressions $\rightarrow$ values
3. Typing rules,
4. which may not be exclusive.

Where do we go from here?

1. Intimate connection between logic and computation: the Curry-Howard isomorphism

Where do we go from here?

1. Intimate connection between logic and computation: the Curry-Howard isomorphism
2. New directions in type theory: guaranteeing security

Where do we go from here?

1. Intimate connection between logic and computation: the Curry-Howard isomorphism
2. New directions in type theory: guaranteeing security
3. New logics and new programming paradigms: linear logic

Where do we go from here?

1. Intimate connection between logic and computation: the Curry-Howard isomorphism
2. New directions in type theory: guaranteeing security
3. New logics and new programming paradigms: linear logic
4. Probabilistic programming languages designed for machine learning

Overview of OCaml

1. Functional - functions are the main entities.
2. Higher-order - functions may take other functions as arguments and
3. may even return functions as results.
4. Typed - every entity has a type.
5. Types are described in their own little language; types are not just the basic types.
6. Expressions may have *multiple* types: polymorphism.

Basic components of any programming language

1. Basic values : `true`, `false`, $1, 2, 3, \ldots$, $1.3, 2.7128$, 'a', 'b'
2. Compound values: data structures,
3. Expressions : an entity that triggers a computation resulting in a value, e.g. $1 + 2 \to 3$.
4. Names : symbols that denote values
5. Bindings : correspondence between name and value established by a definition
6. Parametrized expressions: functions (procedures, methods).

Things we will do without for now

1. Updatable storage - abstraction of memory locations.
2. Control flow - the only control flow will be function applied to an argument.
3. We will incorporate both of these later.

Some concrete examples

1. A binding using the keyword **let**

```
# let x = 1;;
val x : int = 1
# x;;
- : int = 1
```

Some concrete examples

1. A binding using the keyword **let**

```
# let x = 1;;
val x : int = 1
# x;;
- : int = 1
```

2. An evaluation of an expression

```
# let y = x + 1729;;
val y : int = 1730
```

Some concrete examples

1. A binding using the keyword **let**

```
# let x = 1;;
val x : int = 1
# x;;
- : int = 1
```

2. An evaluation of an expression

```
# let y = x + 1729;;
val y : int = 1730
```

3. Function definition and application

```
# let inc = fun n -> n + 1;;
val inc : int -> int = <fun>
# let foo = inc 5;;
val foo : int = 6
```

Recursion

```
# let rec fact n =
  if n = 0 then
    1
  else
    n * fact(n-1);;
        val fact : int -> int = <fun>

# fact 5;;
- : int = 120
```

Thinking recursively

- Do **not** unwind the recursion in your head and trace through the calls and the recursion stack. OK when you are learning for the first time but not the way to think recursively.

Thinking recursively

- Do **not** unwind the recursion in your head and trace through the calls and the recursion stack. OK when you are learning for the first time but not the way to think recursively.
- Three things to keep in mind: (a) exit condition (b) recursive calls must make progress towards the exit condition (c) if the recursive calls are **assumed to work** then check that the body works correctly.

Thinking recursively

- Do **not** unwind the recursion in your head and trace through the calls and the recursion stack. OK when you are learning for the first time but not the way to think recursively.
- Three things to keep in mind: (a) exit condition (b) recursive calls must make progress towards the exit condition (c) if the recursive calls are **assumed to work** then check that the body works correctly.
- NEVER ASK ME TO TRACE THROUGH A RECURSION IN CLASS!!!!!!!!!!!

Tail recursion

- There should be only one recursive call and it should be **outermost.**

Tail recursion

- There should be only one recursive call and it should be **outermost.**
- ```
  # let fastfact n =
    let rec helper(n,m) =
      if n = 0 then m
      else helper(n-1, n * m)
    in
    helper(n,1);;
  # val fastfact : int -> int = <fun>
  ```
  ```

Last example

```
# let even n = (n mod 2) = 0;;
# let odd n = (n mod 2) = 1;;
# let rec rpe base power =
  if base = 0 then 0
  else
    if power = 0 then 1
    else
      if (odd power) then
        base * (rpe base (power - 1))
      else
        let tmp = (rpe base (power/2)) in
        tmp * tmp;;

val rpe : int -> int -> int = <fun>
```