CSC324 Principles of Programming Languages

Lecture 1

September 11, 2019

Welcome to CSC324!

Introduction - Me

Instructor: Lisa Zhang

- **Email**: lczhang [at] cs.toronto.edu
  - Please prefix email subject with 'CSC324'
- **Office hours**:
  - Lisa: Monday 12pm-2pm, DH3078 (and by appointment)
  - Hassan: Tuesday 1:30pm-3pm, CC2110
  - Additional office hours near assignment deadlines and tests.

Introduction - Me

Instructor: Lisa Zhang

- **Email**: lczhang [at] cs.toronto.edu
  - Please prefix email subject with 'CSC324'
- **Office hours**:
  - Lisa: Monday 12pm-2pm, DH3078 (and by appointment)
  - Hassan: Tuesday 1:30pm-3pm, CC2110
  - Additional office hours near assignment deadlines and tests.

Call me (in order of my preference):

- "Lisa", "Prof Zhang", "Prof Lisa", "Prof".

Introduction - You!

Survey – Why are you here?

Raise your hand if you...

- You are taking this course because it is a pre-requisite to CSC384.

Survey – Why are you here?

Raise your hand if you...

- You are taking this course because it is a pre-requisite to CSC384.
- You are taking this course because it is a pre-requisite to CSC488.

Survey – Why are you here?

Raise your hand if you...

- You are taking this course because it is a pre-requisite to CSC384.
- You are taking this course because it is a pre-requisite to CSC488.
- You are taking this course out of interest, and not to fulfill any requirements.

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts. . .

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts. . .
- I am hoping lab attendance isn't required.

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts. . .
- I am hoping lab attendance isn't required.
- Please provide us a lot of practice because it helps us grasp the material better and perform well in the course overall.

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts...
- I am hoping lab attendance isn't required.
- Please provide us a lot of practice because it helps us grasp the material better and perform well in the course overall.
- ...I'm afraid of asking for help...

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts...
- I am hoping lab attendance isn't required.
- Please provide us a lot of practice because it helps us grasp the material better and perform well in the course overall.
- ...I'm afraid of asking for help...
- Make this course easy please

Survey – What you wanted me to know

- I would appreciate it if the lecture slides (if there are any) could be posted sometime before class starts. . .
- I am hoping lab attendance isn't required.
- Please provide us a lot of practice because it helps us grasp the material better and perform well in the course overall.
- . . . I'm afraid of asking for help. . .
- Make this course easy please
- Yeet?

What programming languages have you learned so far?

Imperative Languages

The languages you learned so far follows the **imperative programming** paradigm:

- "Update the value of `x` to 5"
- "Repeat 20 times"

In this paradigm:

- programs are *stateful*
- we perform interesting computations by *manipulating* states

What we will do in CSC324 (Surface Level)

We will learn the programming languages **Racket** and **Haskell**

These languages are **not** typically used in industry .

- . . . though similar languages like Clojure and Scala and are catching on.

But, they are **highly configurable** and very interesting.

- . . . people commonly use Racket (dialect of Lisp) to implement their own languages.

Q: What is a programming language?

Q: Are these programming languages?

- If I write a program that takes two numbers as user input, and adds them, did I create a programming language?

Q: Are these programming languages?

- If I write a program that takes two numbers as user input, and adds them, did I create a programming language?
- If I write a program that takes two numbers and an binary operation ($+$, -, etc) as user input, and apply the operation to the numbers, is *that* a programming language?

Q: Are these programming languages?

- If I write a program that takes two numbers as user input, and adds them, did I create a programming language?
- If I write a program that takes two numbers and an binary operation ($+$, -, etc) as user input, and apply the operation to the numbers, is *that* a programming language?
- What if I also allow users to write functions and save intermediate computations?

Q: Are these programming languages?

- If I write a program that takes two numbers as user input, and adds them, did I create a programming language?
- If I write a program that takes two numbers and an binary operation (+, -, etc) as user input, and apply the operation to the numbers, is *that* a programming language?
- What if I also allow users to write functions and save intermediate computations?
- What if I am building this calculator tool as a web application, with a nice user interface?

What is a programming language?

A programming language is a way of **communicating** a computational task to the computer.

What is a programming language?

A programming language is a way of **communicating** a computational task to the computer.

You can argue that any piece of software is a "programming language" specialized to a particular application domain.

Why should you take CSC324?

Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

. — *Greenspun's tenth rule of programming*

- Softwares you build help users communicate their computational needs
- This course should change the way you think about programming and software design

What we will do in CSC324 (More deeply)

We will define/analyze/modify the **syntactic** and **semantic** features of programming languages, and create small languages of our own.

Specifically, we will:

- Explore **functional programming**, a style of programming that does not allow mutable states
- Explore **macros**, or programs that operate on programs
- Explore various **type systems**
- Explore **logic programming**, where we want to find an answer in a large search space, subject to some constraints (e.g. solve a sudoku puzzle)

. . . and hopefully, expand your definition of programming.

Course Logistics

Evaluation

- 9 exercises (18%; 2% each)
- 2 assignments (22%; 11% each)
- Midterm test (15% or 20%)
- Final exam (40% or 45%)

Exercises

- Exercise due most Tuesdays, 10pm
- Designed to make sure that you keep up with course material
- Completed individually
- No late exercises accepted

Assignments

- Larger/more comprehensive than exercises
- Completed individually or in a group of two
- Grace tokens available
  - Each student starts with 6 grace tokens
  - Each grace token is worth 2 hours
  - Both partners need to have grace tokens to submit late

Midterm Test and Final Exam

We want to give you multiple opportunities to demonstrate what you know!

Whichever gives you the better grade:

- 15% midterm + 45% exam
- 20% midterm + 40% exam

Academic Integrity

- Do not discuss exercise solutions with classmates
- Do not discuss assignments with classmates other than your partner
- Please don't "help" others by helping them solve the homework!
- Please don't put others in awkward positions by asking them for help on an exercise or assignment
- What you can do to help
  - Discuss examples from lecture and the course materials
  - Practice using sample problems
  - Ask questions on the discussion board

Practicals

- Lab attendance is not required for marks
- But it is a great way to practice course material with the support of a TA and other students!
- TAs can help you understand course material and work through examples and previous exercises/assignments
- TAs will introduce some new course material!
- Feel free to bring questions!

Course Notes

- Course notes is written by Prof. David Liu
- The text is free for download on the course website

I recommend reading the notes once *before* lecture.

Course software

- Programming languages
  - Racket 7.4
  - GHC Haskell compiler
- I recommend using the IDE **DrRacket** for Racket
- No particular IDE for Haskell (your choice)
- MarkUs for assignment submission

Course software

We will be primarily using **Racket** in the first half of the course, slowly introducing **Haskell**.

- I will use DrRacket today during lectures
- I might use Jupyter Notebooks in later lectures, to be able to more easily post notes from lecture

Course Languages

- **Racket**
  - Exercises 1-7
  - Assignment 1
- **Haskell**
  - Exercises 4-9
  - Assignment 2

Note Taker

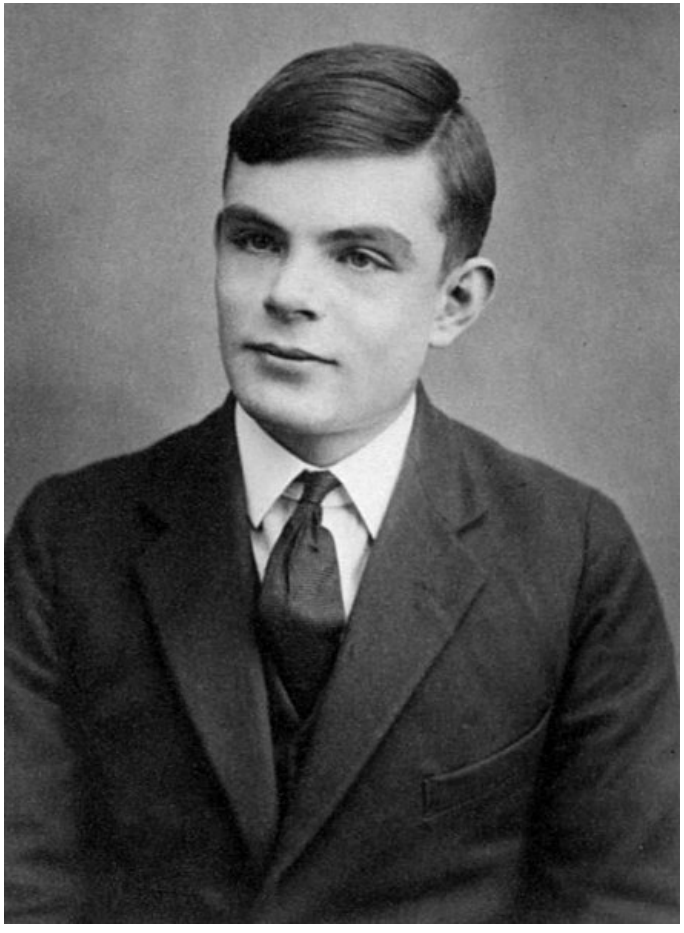
Accessibility Services is looking for reliable volunteers to serve as note-takers this semester.

See http://www.utm.utoronto.ca/accessibility/volunteer-resources/volunteer-note-taker

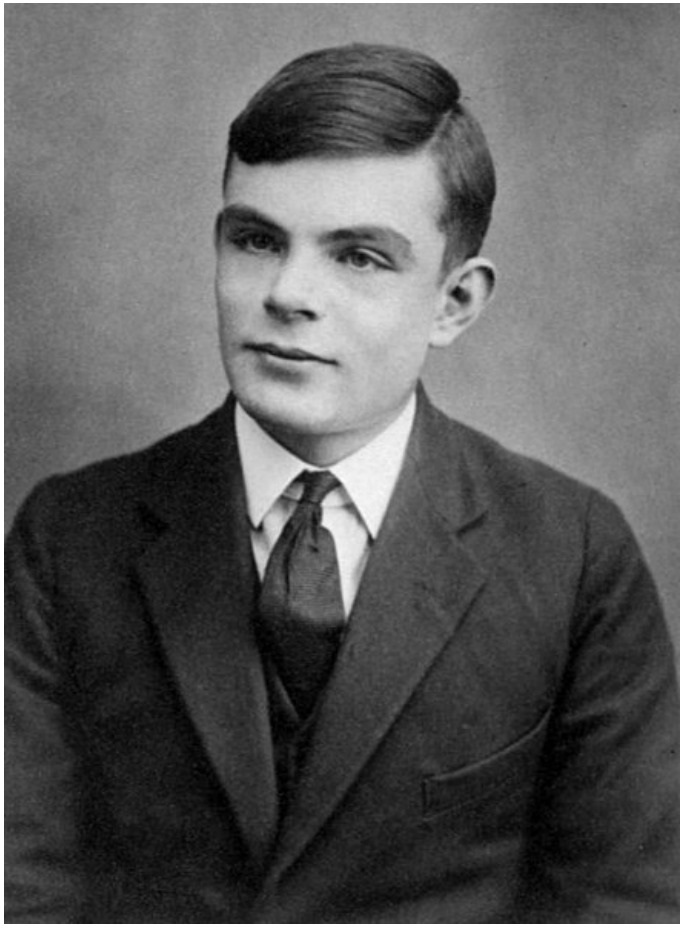
Questions about Logistics?

The Lambda Calculus

Who are these computer scientists?



Who are these computer scientists?



Alan Turing (left), Alonzo Church (right)

Two models of computation

The **Turing machine** performs computation by:

- reading data
- executing instructions to modify internal memory
- producing output

The **lambda-calculus** performs computation by:

- evaluating expressions

Two models of computation

The **Turing machine** performs computation by:

- reading data
- executing instructions to modify internal memory
- producing output

The **lambda-calculus** performs computation by:

- evaluating expressions

It turns out that these two models are equivalent!

Two models of computation

The **Turing machine** performs computation by:

- reading data
- executing instructions to modify internal memory
- producing output

The **lambda-calculus** performs computation by:

- evaluating expressions

It turns out that these two models are equivalent!

Church-Turing Thesis: any other reasonable computational model is equivalent to the Turing machine and lambda-calculus.

The Lambda-Calculus and Functional Programming

The lambda-calculus is the basis of functional programming languages (the "smallest" functional programming language).

Functional Programming: A programming paradigm centered on evaluating functions

To begin our discussion of the lambda-calculus, let's define how we will specify a programming language.

Specifying a Programming Language

Syntax: the structure of valid programs in a language

Semantics: the meaning of the elements of a language

Grammar

Grammar: formal set of rules specifying the syntax of a language

Example: grammar for arthmetic expressions (infix notation)

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Expression

Expression: syntactically-valid element of a language

An expression is syntactically-valid iff it can be *generated* by the language's grammar

Example 1

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- 3

Example 1

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(3 + (1 + 2))`

Example 1

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(3 + (1 + 2))`
- `(3 + 1 + 2)`

Example 1

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(3 + (1 + 2))`
- `(3 + 1 + 2)`
- `3 + 1`

Example 1

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <expr> <op> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(3 + (1 + 2))`
- `(3 + 1 + 2)`
- `3 + 1`
- `1 / 0`

Example 2

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <op> <expr> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- 3

Example 2

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <op> <expr> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(1 + 2)`

Example 2

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <op> <expr> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(1 + 2)`
- `(+ 1 2)`

Example 2

Assuming the following grammar:

```
<expr> = NUMBER
       | '(' <op> <expr> <expr> ')'

<op>   = '+' | '-' | '*' | '/'
```

Q. Are these expressions valid?

- `3`
- `(1 + 2)`
- `(+ 1 2)`
- `(/ (+ 3 4) (- 5 5))`

Example 3

Assuming the following grammar:

```
<list> = null
       | '(' 'cons' NUMBER <list> ')'
```

Q. Are these expressions valid?

- `null`

Example 3

Assuming the following grammar:

```
<list> = null
       | '(' 'cons' NUMBER <list> ')'
```

Q. Are these expressions valid?

- `null`
- `(cons 3 null)`

Example 3

Assuming the following grammar:

```
<list> = null
       | '(' 'cons' NUMBER <list> ')'
```

Q. Are these expressions valid?

- `null`
- `(cons 3 null)`
- `(cons 3 (cons 4 null))`

Example 3

Assuming the following grammar:

```
<list> = null
       | '(' 'cons' NUMBER <list> ')'
```

Q. Are these expressions valid?

- `null`
- `(cons 3 null)`
- `(cons 3 (cons 4 null))`
- `(cons 3 (cons 4 5))`

Example 3

Assuming the following grammar:

```
<list> = null
       | '(' 'cons' NUMBER <list> ')'
```

Q. Are these expressions valid?

- `null`
- `(cons 3 null)`
- `(cons 3 (cons 4 null))`
- `(cons 3 (cons 4 5))`
- `(cons (cons 4 null) (cons 4 null))`

Lambda-Calculus: Syntax

Only three kinds of expressions in the lambda-calculus:

1. Identifier; e.g. x
2. Function expression; e.g. $\lambda x \mapsto x$ (identity function)
3. Function application; e.g. f *expr* (applies f to the expression *expr*)

Lambda-Calculus Grammar

```
<expr> = ID
       | (lambda (ID) <expr>)
       | (<expr> <expr>)
```

(Note: the function expression defines an **anonymous function** – a function with no name!)

Lambda-Calculus: Semantics

- A program in the lambda-calculus is a single **expression** (no **statements**!)
- "Running" such a program means to **evaluate** the expression

Lambda-Calculus Evaluation

- Identifiers and function expressions are already fully-evaluated
- Function applications are evaluated by substituting the argument for the parameter in the body of the function

$$(\lambda x \mapsto x)hi$$

Evaluated by *substituting* *hi* for x in the body of the function, giving *hi* as the result.

How powerful is the lambda-calculus?

Only using functions and identifiers appears restrictive

But, it turns out that . . .

- We can encode numbers, strings, and other data structures
- We can recover **recursion**
- We can perform any computation that you can in any other programming language

Racket

- A much "bigger" language than the lambda-calculus
- Racket is a **functional** language
  - programming paradigm centered on evaluating functions
  - no mutable states

Function Expressions in Racket

```
(lambda (<param> ...) <body>)
```

. . . creates an anonymous function in Racket.

Example:

- `(lambda (x)   (+ x 1))`
- `(lambda (x)   (/ (+ x 2) (- x 2)))`
- `(lambda (x)   (equal? x 2))`
- `(lambda (x)   "hello")`
- `(lambda (x y) #f)`
- `(lambda (x y) (not (equal? x y)))`

Function Applications in Racket

`(<function> <args> ...)`

. . . applys a function to the arguments.

Example:

- `((lambda (x) (+ x 1)) 5)`
- `((lambda (x y) (equal? x y)) #t #f)`
- `((lambda (x y) (equal? x y)) 0 #f)`
- `((lambda (x y) (= x y)) 0 #f)` – gives an error

Pure Functions

```
(lambda (x) (+ x 1))
```

A **pure** function is one whose behaviour is soley determined by the *values* of its parameters, and that returns a value and does nothing else.

- No manipulation of global variables or states
- No reading / writing of files
- No reading / writing to database

Name Bindings

Binding identifiers to values gives us two conveniences:

1. Saving the value of subexpressions.
2. Recursively referring to a function name.

Global Name Bindings in Racket

`(define <id> <expr>)`

Example:

- `(define x 5)`
- `(define add1 (lambda (x) (+ x 1)))`

Alternative syntax for binding functions:

- `(define (add1 x) (+ x 1))`

Local Name Bindings in Racket

```
(let* ([<id> <expr>] ...) <body>)
```

Example:

```
(define (f x y)
  (let* ([e1 (+ x 1)]
         [e2 (- y 2)])
        (* e1 e2)))
```

(Note: There are subtle differences between `let`, `let*` and `letrec`)

Referential Transparency

Identifier is **referentially transparent** if it can be substituted by its value without changing the meaning of the program.

For this reason, **an identifier cannot be bound more than once**!

Racket Symbols

A **symbol** is a quoted name

Example:

- `'a`
- `'name`
- `'symbol`

Symbols are **not strings**, and *cannot* be decomposed into characters!

Lists in Racket

In Racket, a list is really a **linked list**, with a first-node and a reference to the rest of the list.

Example:

- `null` is the empty list
- `(cons 1 null)` is the list containing 1
- `(cons 1 (cons 2 (cons 3 null)))` is the list containing 1, 2, 3
- `(list 1 2 3)` is also the list containing 1, 2, 3
- `(list 'a 2 #f)` is the list containing `'a`, 2, `#f` – list elements can be different types

List Quoting

- `'(a b c)` is the list containing the symbols `'a`, `'b`, and `'c`
- `'(1 2 3)` is the list containing the numbers 1, 2, 3
- `'((a b) (1 2 3))` is a two element list, where:
  - the first list contains the symbols `'a`, `'b`
  - the second list contains the numbers 1, 2, 3
- `'()` is the empty list

List Unpacking

```
> (define lst (list 1 2 3))
> (first lst)
1
> (rest lst)
'(2 3)
```

Recursion on Lists

- We can operate on the first element of a list, and then recurse on the rest

```
(define (add1-list lst)
  (if (null? lst)
    (list)
    (cons (+ 1 (first lst))
          (add1-list (rest lst)))))
```