CS 360: Programming Languages
Lecture 1: Introduction to Programming Languages

Geoffrey Mainland

Drexel University

Section 1

Administrivia

Administrivia

- Instructor: Geoffrey Mainland
  - E-mail: mainland@drexel.edu
  - Office: University Crossings 106
  - Office hours (for now): Mondays 4pm–7pm; Thursdays 5pm–6pm. You may also schedule another time via Piazza. **Office hours are blocks of time reserved for you**.
- Teaching Assistant: Xiao Han.
  - In-person office hours will be held in the Cyber Learning Center Wednesday 2pm–4pm and Thursday 12pm–2pm.

You are responsible for reading the material posted on the course web site. In particular, read the syllabus.
https://www.cs.drexel.edu/~mainland/courses/cs360-201825/

Lectures

- All sections will be recorded. You may access recordings on BlackBoard.
- I do keep track of who shows up to class.
- This is not an online class!

Grading policy

Grades will be weighted as follows:

- Homework, labs, and participation (60%) (Piazza participation counts).
- Midterm (20%)
- Final Exam (cumulative) (20%)

The course will be curved (in your favor).
Late work:

- All students have two late days, which may be used at any time to turn in homework late without penalty. You must notify us (on Piazza) that you are using a late day **before the deadline**.
- After consuming all late days, the late penalty is 25% per day per homework.
- In the case of extenuating circumstances, please contact me and we will make arrangements.

Turning in homework

- All work will be submitted using GitHub Classroom. Homework 0 covers the details on how to use GitHub for assignments. **Get your GitHub account now**.
- Homework will typically be due Mondays at 11:59PM Eastern. I have office hours Monday afternoon/evening from 4pm–7pm.
- The first homework, Homework 0, is due **this Friday, January 11**. It's easy—all you have to do is accept the assignment and push a file to your homework 0 GitHub repository. **Get your GitHub account ASAP**.
- **Make sure you connect your GitHub account to your Drexel account!** Follow the instructions in Homework 0.
- Homework 1 is due **Monday, January 14** (it is also relatively easy).
- Read over both of these assignments **after class** so you know what you're in for.
- We provide TravisCI support for testing your assignments. I recommend you get that working too (see Homework 0).

Poll regarding homework deadline

I will run a Piazza poll **this week** to give you the option to choose a different deadline day for **future** homeworks (deadlines for homeworks 0 and 1 will not change).

Course Communication

- Please use Piazza for (almost) all course communication.
- If you need to contact me about a personal issue (family emergency, scheduling a meeting time, etc.), you are of course welcome to email me directly. You may also send me a private message on Piazza.
- All communication about course **content** should be via Piazza.
- Please **do not** post solutions to the homework, partial or otherwise. If you are unsure whether or not a post should be public, send it privately to the instructors on Piazza.
- If you have a course-related problem I might be able to help with, let me know **ASAP**. Don't wait until after a deadline, or, worse yet, until after the term has ended!

Course Software

- We will be using Racket 7.1 and GHC 8.4.3.
- All software is installed on tux, although you need to read our instructions to ensure you are using the correct versions—see the course home page.
- We are providing a course virtual machine. If you are using the course virtual machine, all software is ready out of the box. The course home page has instructions for downloading the VM.
- The virtual machine has DrRacket installed, which is a much nicer environment for writing Racket code than the tux command line.
- You may use whatever environment you are comfortable with, but **you are responsible for making sure your code works on tux with the software versions we specify**.

Academic Honesty

- You must adhere to Drexel's policy on academic honesty.
- All work must be your own.
- You may not use code from others or from online *unless you receive explicit permission to do so*. You *may* use code we give you.
- **You may not show your code to others.**

What makes a good language?

```
[] == ![];

true == [];
true == ![];

document.all instanceof Object;
typeof document.all;
```

What makes a good language?

```
[] == ![]; // -> true

true == []; // -> false
true == ![]; // -> false

document.all instanceof Object; // -> true
typeof document.all; // -> 'undefined'
```

It's not just JavaScript…

$$x + y - x = \ldots$$

$$x + y - x = y$$

$$x + y - x = 0 \quad \text{for } x \neq 0$$

Course topics

- Models of computation.
- Functional programming (Scheme and Haskell).
- Polymorphism.
- Semantics.
- Types; algebraic data types.
- Proving properties of programs.
- Property-based testing.
- Regular expressions (lexing) and grammars (parsing) (maybe).

Who cares?

- Even if you never use these languages “in the real world,” being exposed to them should help you think more clearly about the code you write *no matter what language you are using*.
- Having a CS degree should mean that you can pick up a new language quickly—this course will help you acquire that skill.
- A good programmer thinks in terms of *invariants*. This course will encourage you to think is this way, in particular by focusing on program *semantics*.
- Even if you never have to muck about “under the hood” of the language (or tool) you’re using, it’s good to have some idea of how the bits fit together (why?).
- Many previously “academic” language features have made their way into languages used in industry—good bet that this will continue to happen in the future.

Section 2

What is PL?

Describing a language

- *Syntax*: the rules that define the combination of symbols that represent valid programs.
- *Semantics*: defines the “meaning” of a program.

What is “programming languages?” (PL)

- The field of programming languages is concerned with *reasoning about programs*.
- To reason about programs, we need a language *semantics*.
- Encompasses language design as well as program analysis.
- We will also look at aspects of language implementation.

Language Paradigms

- **Imperative**: sequential execution; variables represent memory locations; use of assignment to change values of variables.
- **Object-oriented**: typically an extension of the imperative paradigm, but programs are constructed from small, modular pieces.
- **Functional**: based on the lambda calculus; functions are first-class and may be functions in the mathematical sense.
- **Logic**: based on symbolic logic.
- We will focus on functional programming...

Section 3

Models of Computation

Turing Machines: Informally

- A finite **alphabet** consisting of a special **blank** symbol and one or more other symbols.
- An infinite **tape** divided into **cells**, each of which contains a symbol from the alphabet. All but finitely many cells start out containing the blank symbol.
- A **head** that can read and write symbols on the current cell and move left or right one cell at a time.
- A **state register** that stores the state of the Turing machine. There are finitely many states and one special state, the **start state**, which is the initial state contained in the state register. Additionally, there is a set of **accepting states** that say when the TM is "done."
- A (finite) **table** of instructions. Given the current state and current symbol (under the head), the instruction tells the machine to do the following ("in order"):
    - To either erase or write a symbol in the current cell.
    - Move the head either left or right.
    - Assume a new state.

Turing Machines: Formally

A Turing machine[1] is a 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ where

- Q is a finite, non-empty set of **states**.
- $\Sigma \subseteq \Gamma \setminus \{B\}$ is the set of **input symbols**.
- Γ is a finite, non-empty set of **tape symbols**.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times \{L, R\}$ is a partial function, called the **transition function**, where L is left shift and R is right shift.
- $q_0 \in Q$ is the initial state.
- $B \in \Gamma$ is the **blank symbol**. The blank symbol initially appears in all but the finite number of initial cells that hold input symbols.
- $F \subseteq Q$ is the set of **accepting states**.

Questions:

- What does $\Gamma \setminus \{B\}$ mean?
- What is a partial function?

[1] See [HMU06]

Church-Turing Thesis

Anything that is computable in an informal sense can be computed by a Turing machine.

Turing Machines

- Turing Machines are a staple of undergraduate computer science, and if you have not seen them yet, I assume you will see them again at some point—just not in this class.
- As a model of computation, they should remind you of something you use every day. . .
- We have a different tool at our disposal. . .

The λ-Calculus

- The lambda calculus is a language of pure functions.
- Serves as the basis for most functional languages (Scheme, ML, Haskell, F#, Scala, Clojure, etc.).
- Interestingly, it *is a myth* that Lisp was originally based on the lambda calculus [Tur12].
- Lambda calculus is all about *functions*. In fact, all it has is functions!
- But are functions enough?

λ-Calculus Syntax

$$\begin{array}{lcll} t & ::= & x & \text{variable} \\ & | & \lambda x.t & \text{abstraction (function)} \\ & | & t\ t & \text{application} \end{array}$$

- This is a **grammar** for the lambda calculus.
- There are three different ways to construct a lambda calculus **term**, represented by t.
- The t seen in the abstraction and application cases represents *any* term that can be generated from the grammar.

λ-Calculus Syntax

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & \mid & \lambda x.t & \text{abstraction (function)} \\ & \mid & t\ t & \text{application} \end{array}$$

- Think of $\lambda x.t$ as a function; the function takes a single argument x, and it may then use x in the **body** of the function, t.
- How can we write the identity function? $\lambda x.x$
- Application is left-associative, so $x\ y\ z$ is equivalent to $(x\ y)\ z$.
- The body of a lambda extends as far to the right as possible, so

  $$\lambda x.x\ \lambda z.x\ z\ x$$

  is equivalent to

  $$\lambda x.(x\ \lambda z.(x\ z\ x))$$

λ-Calculus terminology

$$\lambda x.x\ y$$

- The **scope** of x is the entire body of the function.
- The variable x is **bound** in this term.
- The variable y is **free** in this term.

In Scheme:

```
(lambda (x) (x y))
```

Scope and Shadowing

$$\lambda x.\lambda x.x\ x$$

- The outer x is not available in the body of the function because it is **shadowed** by the x bound by the inner lambda.
- To refer to the the outer x, give the two x's different names: $\lambda x.\lambda y.x\ y$

In Scheme:

```
(lambda (x) (lambda (x) (x x)))
```

α-Equivalence

- *Consistently* renaming variables doesn't change the semantics of a term.
- Two terms that differ only in the names of bound variables are called α-**equivalent**.
- $\lambda x.x \equiv \lambda y.y \equiv \lambda z.z$
- $\lambda x.\lambda y.x\ y \equiv \lambda y.\lambda x.y\ x \equiv \lambda p.\lambda q.p\ q$

In Scheme, compare this:

```
(lambda (x) x)
```

to this:

```
(lambda (y) y)
```

Semantics

There are several ways to use mathematics to give a precise meaning—that is, a semantics—to programs.

- **Operational Semantics:** the behavior of a program is specified in terms of the operation of an abstract machine. The abstract machine uses terms of the language as its "machine code." The machine has a **state**, which for simple languages is just a term in the language, and a **transition function** that specifies how to move from one state to the next. The *meaning* of the program is the final state of the abstract machine.
- **Denotational Semantics:** the meaning of a program is taken to be some mathematical object. This requires defining **semantic domains** and an **interpretation function** that maps terms into elements of these domains.
- **Axiomatic Semantics:** describes the behavior of a program in terms of logical preconditions and postconditions.

We will look at an operational semantics for the λ-calculus.

Inference rules

$$\frac{P \qquad P \to Q}{Q} \qquad (\textsc{Modus ponens})$$

- Inference rules consist of one or more **premises** and a **conclusion**.
- Premises occur above the line and conclusions below the line.
- Rules without a premise are called **axioms**.
- You used inference rules in CS 270 to find the “meaning”, i.e., semantics, of Boolean formulas.
- We will use inference rules to define a semantics for the λ-calculus.

λ-Calculus: Operational Semantics

$$\begin{array}{rcll} t & ::= & x & \text{variable} \\ & | & \lambda x.t & \text{abstraction (function)} \\ & | & t\ t & \text{application} \end{array}$$

$$v ::= \lambda x.t$$

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \quad (\text{E-App1})$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \quad (\text{E-App2})$$

$$(\lambda x.t)\ v \longrightarrow [x \mapsto v]t \quad (\text{E-AppAbs})$$

- Left-hand column defines syntax and **values**. Values are terms that have been fully evaluated, i.e., terms in their "final" form.
- Right-hand column defines an **evaluation relation** $t \longrightarrow t'$, pronounced "t evaluates to t' in one step."
- This is an **operational semantics**.
- To figure out which rule(s) apply, match the syntactic form in the conclusion to the term you are attempting to evaluate.
- Question: Why might we call this an *evaluation relation* rather than an *evaluation function*?

λ-Calculus: Operational Semantics

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2} \qquad (\textsc{E-App1})$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'} \qquad (\textsc{E-App2})$$

$$(\lambda x.t)\ v \longrightarrow [x \mapsto v]t \qquad (\textsc{E-AppAbs})$$

- Only three rules—that's all you need to know to evaluate expressions in the λ-calculus!
- What is the order of evaluation?

λ-Calculus: Example

$$\frac{t_1 \longrightarrow t_1'}{t_1\ t_2 \longrightarrow t_1'\ t_2}\ (\text{E-App1}) \qquad (\lambda x.t)\ v \longrightarrow [x \mapsto v]t\ \ (\text{E-AppAbs})$$

$$\frac{t_2 \longrightarrow t_2'}{v_1\ t_2 \longrightarrow v_1\ t_2'}\ (\text{E-App2})$$

- Evaluate: $((\lambda x.\lambda y.x\ y)(\lambda w.w))(\lambda z.z)$
- First step:

$$\frac{(\lambda x.\lambda y.x\ y)(\lambda w.w) \longrightarrow \lambda y.(\lambda w.w)\ y}{((\lambda x.\lambda y.x\ y)(\lambda w.w))(\lambda z.z) \longrightarrow (\lambda y.(\lambda w.w)\ y)(\lambda z.z)}\ \text{E-App1}$$

- Second step:

$$\frac{}{(\lambda y.(\lambda w.w)\ y)(\lambda z.z) \longrightarrow (\lambda w.w)(\lambda z.z)}\ \text{E-AppAbs}$$

- What next?

λ-Calculus: Another example

$$\begin{aligned} & (\lambda x.x\ x)\ (\lambda x.x\ x) \\ \longrightarrow & [x \mapsto \lambda x.x\ x]x\ x \\ \equiv & (\lambda x.x\ x)\ (\lambda x.x\ x) \end{aligned}$$

- Easy to write non-terminating computations.
- What else can we do with the λ-calculus?

Representing Numbers Using Functions

- Remember the question we asked: are functions enough?
- Big idea: a “number” *n* **is a function** that takes a function and a value and applies the function to the value “*n* times.”
- *n* is $\underbrace{\texttt{(lambda (f x) (f (f (... (f x)))))}}_{n \text{ applications of } \texttt{f}}$
- Let’s implement this idea in Scheme.

Church Encodings

- These are the **Church numerals**.
- Representing a data type and associated operations in this way is a **Church encoding**.
- We can do this for *any* data type.
- Gives us some intuition as to why the lambda calculus is sufficient for representing any computation. Church proved that its expressive power is equivalent to that of Turing machines.

Expressive power of the λ-calculus

- Actually, we can do "everything." And usually much more pleasantly than a Turing machine.
- High-level operations can be translated into the lambda calculus.
- A *very* important principle in programming languages: define a small core language, and then show how more complex constructs are translated into the core language.
- We then reason about complex features in terms of their translations.
- Fewer cases, simpler to reason about.

Models of Computation

- Thinking back to the language paradigms we saw at the beginning of lecture, which paradigm does each model of computation remind you of?
- Do these models aid “code” reuse? Encourage abstraction?
- Which would you use to reason about program correctness?
- Which would you use to reason about program efficiency?
- Can you think of a cost model for Turing machines? How about the lambda calculus? Do these cost models reflect the cost of operations on a “real” computer?

Section 4

References

References

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Prentice Hall, Boston, 3rd edition, July 2006.

[Tur12] D. A. Turner. Some history of functional programming languages. In Hans-Wolfgang Loidl and Ricardo Peña, editors, *13th International Symposium on Trends in Functional Programming (TFP '12)*, number 7829 in Lecture Notes in Computer Science, pages 1–20. Springer Berlin Heidelberg, St. Andrews, UK, June 2012.