Comp 311
Functional Programming
Lecture 1

Robert “Corky” Cartwright
Rice University

Robert “Corky” Cartwright

- PhD, semantics and verification of (first–order) functional programs
  - Stanford 1976[1977]
    - Official Advisor: David Luckham
    - Primary Mentor: John McCarthy

- 45+ years of Computer Science research
  - PL theory
    - First-order Programming Logic
    - Semantics of types, sequential functional languages
    - Type systems (Soft typing)

  - PL systems (software engineering)
    - Soft type checker for Scheme
    - Testing concurrent programs
    - DrJava including Functional Java Subset

Course Overview I

- An Introduction to Functional Programming
- Lectures: Tuesdays and Thursdays: 9:25am – 10:40am
- Office hours: Corky
  - Duncan Hall 3104
  - Tuesdays and Thursdays 1:30pm – 2:30pm
  - By appointment

Course Mechanics

- Course website: are https://github.com/JavaPLT/Comp-311-Fall-2021
- Former course websites: https://comp311.rice.edu
  - Syllabus and lectures posted here
  - Lecture topics are subject to change
- Piazza: https://piazza.com/rice/fall2021/comp311
  - Course announcements and Q&A forum
  - Homework assignments and practice exams posted here
- Grading
  - 50% Homework Assignments
  - 25% Mid-term
  - 25% Final
  - Extra credit points on exams, some assignments

Course Overview II

- No required textbook purchase
    - We will draw from a variety of sources including free online textbooks and monographs. Some of them are available for purchase in printed form from online bookstores if you choose.

- Coursework consists primarily of weekly homework assignments that are either short programming assignments or written assignments about the underlying theory.

- Make sure you do these assignments! They embody the key ideas and principles covered in the course.

Course Culture

- Basic course on functional programming
    - With the possible exception of the material on Haskell at the end of the course, the content should be accessible to freshmen with little background in Computer Science who know the rudiments of Java, *e.g.*, have taken an AP Computer Science course that covers Java programing.
- Coursework consists primarily of weekly homework assignments that are either short programming assignments or written assignments about the underlying theory.
- Make sure you do these assignments! They embody the key ideas and principles covered in the course.
- Why functional programming matters (expounded on next slide)

Two basic models of computation

- **Mutate state**. Example: simulating a machine language program for a OTS x86 processor. Too messy to illustrate.

- **Simplify a symbolic expression**. Example: reducing an arithmetic expression to a value, *e.g.*,

$$(17 + 5) - 3 \;=\; 22 - 3 \;=\; 19$$

Read: [whyfp.pdf (chalmers.se)](https://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf)
[https://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf]

Mutating State

- In this model, a computation is a sequence of machine states that begins with a start state and generates a potentially infinite sequence of steps where each step is a new state completely determined by the preceding state.
  - The state includes both the program and the data being processed by the program.
  - The state space is infinite if the model is Turing-*complete* (capable of implementing any computable function).
  - The simplest such model is Turing machines, but modern so-called Random Access Machines (RAMs) such as x86 processors that are ubiquitous (billions of microprocessors!) embody the same basic model, except that they impose size limits on the allowed states. Of course this limit increases if you add more memory (including auxiliary storage) to the machine.
  - Each step in a computation *updates* the state of a machine.
  - A primary advantage of this model is that it makes the costs of computations manifest by counting state-change steps (or equivalently the number of states appearing in a computation that terminates).
  - A disadvantage is that even trivial computation are messy and full of tedious details. Hence, examples are very time-consuming to present so I won't.

Simplifying Expressions

In this model (sometimes called the *reduction* model", a computation is a potentially infinite sequence of "reduction steps" that that transform a starting expression to an irreducible expression that is called the "answer". Each reduction step simply replaces a sub-expression of an expression by an equivalent expression that is closer to being an answer.

- Simplification replaces a *program* by a "simpler" *program.*
- This computational model is actually *more familiar* to most students because we all learn how to do arithmetic in grammar school. Evaluating the arithmetic expression

  $$(17 + 5) - 3 = 22 - 3 = 19$$

  is a very simple computation in this model.
- In the simplest version of this model, every expression is a tree constructed from a countable collection of primitive operations of fixed arity (including constants).
- The linear expression

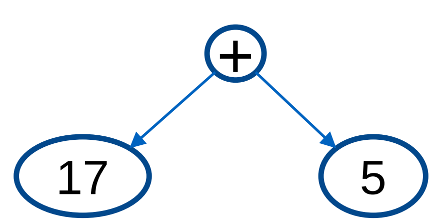
  $$(17 + 5) - 3$$

  is really a tree with the − operation at the root. In linear symbolic notation, we typically encode the structure of the tree using parentheses.

Multitude of different syntactic definitions of expressions

- You are undoubtedly familiar with some of these definitions as in Python, Pascal, Java, C, C++, Scala, Swift, Rust, Cotlin, Haskell, JavaScript, Perl, Snobol, APL, COBOL, Fortran, Modula, Cedar, … , Racket/Scheme. If we ignore tedious differences in syntactic conventions and outright pathologies (*e.g.*, APL) , there typically are only a few important differences:
  - Are variables allowed and can they be bound inside expressions? (Necessary for Turing completeness?)
  - Can compound expressions appear as operators? (Often stated: are functions "first-class" data values) The answer regrettably is "No" in most mainstream languages (e.g. Pascal, Java, C, C++, JavaScript, …). Even first-order logic bans such constructions. (Unnecessary for Turing completeness; explicit *apply* operator is a common hack.)
  - Many languages that do not explicitly support "functions as first class values" have tolerable workarounds like anonymous inner classes in Java (now abbreviated by "lambda expressions" [Church is now turning in his grave]).

The intuitive key to mastering the "reduction" model of computation is to think of expressions as trees, often called *abstract syntax* trees (ASTs). In such a tree every operator is the root of a subtree where its operands (arguments), represented as ASTs, are its children. Hence, (17 + 5) is represented by the AST



Homework Assignments

Think of the programming assignments in this class as very short essays. Focus as much on style as you would for an essay.

50% of a homework grade is based on clarity and style

50% on correctness

Homework Assignments

- Projects are due one week after being assigned.
- Each student has 7 “slip days” to address scheduling conflicts and minor sickness. No more than 3 “slip days” can be used on a given assignment unless you get explicit permission from the instructor.
- Hoard your slip days. The assignments will be progressively more challenging. I predict that some students will not use any slip days.
- Expect to spend about 10 hours outside of class per week.
- Block this time off now in your schedule and respect these commitments.

Homework Assignments

- Assignments are published on Thursdays.
- Start on assignments early so that you have time to ask questions in class, on Piazza, and at office hours.
- A positive attitude and tackling assignments early will help you do your best in the course.

Homework Assignments

- All assignments will be small in scale.
- Most will be given in (the functional subset of) Racket which is a very simple, pure functional language that is easy to simulate in modern type-safe languages like Java, C#, SmallTalk, and Javascript (particularly TypeScript). We will document Racket programs with types. There is an advanced Racket with a “gradual” type system but I disagree with the details if not the spirit.
- We will show how to simulate functional programming in Java, exposing most of the technology used to implement Scala (and perhaps Swift).
- We will also write some programs in Haskell so you are well-equipped to use it as a software engineer.

Homework Assignments

- I strongly recommend that you use the DrRacket programming environment to develop and test Racket programs. The Racket platform runs on Windows, MacOSX, and Linux. If you have a Chromebook, I suggest that you run Linux on it.
- For Java, you have the option of using DrJava or a professional IDE like IntelliJ IDEA or Eclipse but I only use DrJava so I won't be able to answer questions about the professional IDEs.
- I am still researching Haskell platforms but I am leaning toward Visual Studio Code plus a few plugins. IntelliJ is another possibility.
- We will use SVN (turnin on CLEAR) for all assignments.
- Instructions on how to format and submit assignments will be posted on the course github website:

    github.com/JavaPLT/Comp-311-Fall-2021.

What is Functional Programming?

Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- The Lambda Calculus (Church)
- Post Machines (Post)

The creators of these models were surprised when they all turned out to be equivalent if computations are confined to functions mapping finite inputs to finite outputs. Now we understand that the notion of computability is an utterly fundamental notion in mathematics.

With exception of Lambda Calculus, all of these models are "bottom-up" frameworks for pushing bits or symbols. But even the Lambda Calculus had a grubby syntactic character because there was no model based on defining and applying functions. It was a vision, an intuition until Scott supplied a truly functional model that could handle self-application and support an isomorphism between D and D -> D.

Early Models of Computation

- Turing Machines (Turing)
- Type-0 Grammars (Chomsky)
- **The Lambda Calculus (Church)**
- … *and many others*
- To the surprise of their inventors, all of these systems turned out to be equivalent in expressive power.
- Suggests there is a deeper structure to the nature of computation.

The Lambda Calculus

- A *calculus* consists of a set of rules for rewriting symbols.
- An attempt to rebuild all of mathematics on the notion of *functions* and *applications.*
- There is no mutation in the lambda calculus; it is a reduction system.
- Every program consists solely of applications of functions to arguments (which are also functions in the pure lambda calculus, a misleading restriction IMO)
- Applications of functions return values (which are also functions)
- Encoding numbers as functions does not work out well; in the pure lambda calculus, numbers are actually encoded as syntactic descriptions of functions. Equality of functions is undecidable.
- The Pure Lambda Calculus was a critical step in the right direction but it was NOT a true functional programming language. If you add a few constants (*nats*, *suc*, and *if-zero* conditional expressions), you get PCF which is a true universal functional programming language. But even PCF is incomplete in fundamental (if practically unimportant) ways.

What is Functional Programming?

Every program is a collection of function definitions plus an execution expression. For example, assuming our programming language includes the natural numbers, booleans, and a few simple primitive operations on natural numbers and boolean, we can define a program for computing factorial is as follows:

$$\text{fact } 0 = 1$$
$$\text{fact } n{+}1 = n \times \text{fact}(n - 1)$$

and compute fact(1000)

Why Avoid Side Effects?

- **Programs are easier to write:** There are fewer interactions between program components, enabling multiple programmers (or a single programmer on multiple days) to work together more easily. Moreover, essentially all data types have simple inductive definitions which provide a simple framework (recursive defintions of functions) for writing code.

- **Programs are easier to read:** Pieces of a program can be read and understood in isolation.

- **Programs are easier to test:** Less context needs to be built up before calling a function to test it.

- **Programs are easier to debug:** Problems can be isolated more easily, and behavior is inherently deterministic and **local**.

- **Programs are easier to reason about:** The model of computation needed to understand a program without mutation is much simpler; it is ordinary algebra plus induction on the structure of the data.

Why Avoid Side Effects?

- **Programs are easier to execute in parallel:** Because separate pieces of a computation do not interact, it is easy to compute them on separate processors

- This is an increasingly important consideration in the era of multicore chips, big data, and distributing computing

  - *This advantage undermines an often cited argument for mutation (efficiency)*

What is Functional Programming?

- A style of programming that emphasizes functions as the basis of computation
  - Functions are applied to arguments
  - Functions may be passed as arguments to other functions
  - Functions may be returned as values of applications

Pause!

Why Emphasize Functions?

- Functions allow us to factor out common code
  - DRY: Don’t Repeat Yourself
  - Why is DRY important?
    - Program understanding
    - Program maintenance
  - Passing functions as arguments is often the most straightforward way to abide by DRY
  - Returning functions as values is also important for DRY

Why Emphasize Functions?

- Functions allow us to concisely package computations and move them from one control point to another
- Aids us with implementing and reasoning about parallel and distributed programming (yet again)
- Reasoning about sequential programs is easier

    Equational reasoning + induction

A Word on Object-Oriented Programming

- There is no tension between functional and object-oriented programming. In fact, OOP can be cast as an enrichment of FP. See https://www.cs.rice.edu/~javaplt/papers/OOPEnrichesFP.pdf

- In many ways, they complement one another.

- Languages like Scala and Swift are designed to integrate both styles of programming

Pause!

Quick Start with Racket

To install Racket on Windows, MacOsX, or Linux,

- Go to https://racket-lang.org/download/ and download the “regular” version of Racket.
- Execute the downloaded installation file.
- Play with Racket arithmetic and simple functions on numbers. Racket performs rational arithmetic until forced to use inexact approximations.

Generalized Expressions

- Simple expressions are powerful in principle but clumsy to use in practice. (This issue is explored in detail in Comp 411.) We can generalize simple expressions and view all operators as atomic expressions, not just operators of arity 0. Simple expressions are built-up inductively using a countable collection of operators. Each operator has a fixed associated arity, a natural number specifying how many operands (arguments) it takes. To get this inductive process started we need some operators with arity 0. In very simple languages, we often call such operators *constants*. For example, in the arithmetic expression

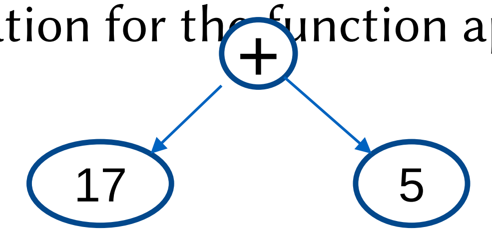
  $(17 + 5) - 3$

  the numbers 17, 5, and 3 are constants. The operators + and − have arity 2 and in standard mathematical notation appear in "infix" form. In infix notation, the operands of an operator appear before and after the operator as in

  $17 + 5.$

  In formal mathematical logic, the notation +(17,5) is typically used instead, to support a uniform syntax for the applications of all operators with arity > 0.

  Conceptually, the best representation for the function application 17 + 5 is an expression tree



where the root is the operator (in this case +) and the children are trees representing the operand expressions (in this case **17** and **5**). In such a tree, all operators including constants are tree nodes and every operator node of arity $k > 0$ has k subtrees representing the operands. There is a one-to-one correspondence between fully parenthesized prefix notation (since the operator appears immediately before it operands) and tree notation for simple expressions.