## CS345H: Programming Languages

## Lecture 1: Introduction and Lambda Calculus I

Thomas Dillig



## What is this course about?

- This course is about programming languages
- We will study different ways of specifying programs
- We will learn how to give (precise) meaning to programs
- We will see how to use programming languages to prevent run-time errors
- We will explore these concepts in real-world languages



## Why should you take this course?

- Understanding programming languages means that you will be able to program in any existing or future programming language almost immediately
- You will be able to choose the right language for the right problem
- You will have techniques to give precise semantics to any string, not just programs.
- You will have a much easier time getting (and keeping) jobs ;-)



## Course Administration

- (Tentative) syllabus is on class website at cs.utexas.edu/~tdillig/cs345h
- Instructor: Prof. Thomas Dillig
- TAs: Pengxiang Cheng
- Office hours: See course website for updates
- We also use Piazza
- Check this website and Piazza!



## Course Administration

This class has the following requirements:

- You will build an interpreter for a realistic language.
- Substantial project, but broken up into 4 manageable programming assignments
- One larger, open-ended project
- We will have approx. weekly written homeworks
- Two in-class midterms and final during finals week.
- This is a difficult class with a substantial workload



## Course Administration - Dates

The following exams are scheduled:

- Midterm 1: 10/11 in class
- Midterm 2: 11/15 in class
- Final: 12/11 in class
- You must be available at these dates, no alternate exams.
- If you miss an exam, your score is 0.

## Grading

- Grades breakdown
  - 15%: each midterm
  - 25% Final
  - 20% Written Assignments
  - 25% Programming Assignments
- Each written assignment is due at the beginning of class, each programming assignment at midnight on the due date.
- You have 3 24-hour period late days to use, but you cannot use more than 2 late days on one assignment.
- Anything handed in after this will receive 0 credit.



## Grading

- The final grades will be curved
- However: Your grade will never get worse from curving, only better
- You will receive lots of feedback through assignments and midterms
- We will post average and standard deviations on all scores, so you know how you are doing



## Getting Help

- We will use the newsgroup function in Piazza for any questions about homework, programming assignments and material.
- We will not answer any emails about these topics
- For any personal issues reach out directly to me via email.



## Collaboration

- You must complete the written assignments individually
- If you discuss the assignment with other students, you must acknowledge their names on your assignment
- You may complete the programming assignments alone or in pairs; you can change your parter on each project, but not during one project
- We use plagiarism-detection software to ensure your programs are not copied. Any cheating will result in an F for the course and referral to the UT honor code violation committee



## Other Policies

Some comments:

- No makeup anything to improve grades
- Grades are final, I will never change the course grade after the semester
- It is your responsibility to check for grading mistakes on Canvas when assignments are handed back. If we don't hear from you within a week, your score is final
- You are responsible for anything announced in class



## Let's get started!

## History of Programming Languages

- It all started in 1954, with the IBM 704 computer



## History of Programming Languages

- This computer was programmed with assembly instructions written on punch cards
- Problem: For the first time in IBM's history, software development costs exceeded hardware cost!
- Solution proposed: Program computer in a higher-level language than assembly



## FORTRAN I

- Enter John Backus
- Translation from higher-level language to assembly had already been tried before...
- And did not work out (at all)
- But team lead by John Backus produced first practical programming language called FORTRAN and a compiler to translate it to assembly



## Impact of FORTRAN

- Within 2 years: 80% of programs written for the IBM 704 were written in FORTRAN
- This is even though FORTRAN I is a pretty awful language (by today's standards)
- After this: Almost all programming done in (increasingly) higher level languages
- Programming languages have greatly improved programmer productivity, enabling software that would never haver been possible otherwise

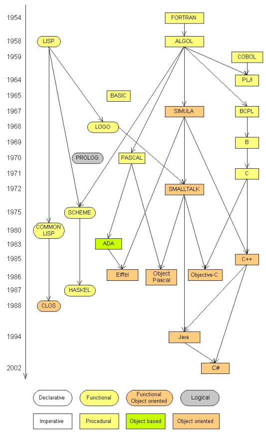


## Language Goals:

- In the beginning, overarching concern when developing languages was performance
- As hardware got faster, many different goals emerged: Reliability, Security, Ease of Use, Re-usability, etc
- This resulted in thousands of actual programming languages



## Language Evolution



## Language Design Today

- We understand pretty well how to design good programming languages
- However, many bad languages are still designed
- After this class, you will be able to recognize bad programming languages



## Lambda Calculus

- There are many programming languages we could talk about
- But pretty much all real languages are complex, large and obscure many important issues in irrelevant details
- We want: "as simple as possible" language to study properties of programming languages
- This language is known as lambda calculus



## Lambda Calculus

- There are only four expressions in lambda calcus:
- Expression 1: constants
  - 1, 7, "yourName" are all valid expressions in lambda calculus
- Expression 2: identifiers
  - Will usually use x, y, etc for those
- Expression 3: lambda abstraction
  - written as $\lambda x.e$
- Expression 4: application
  - written as $e_1\ e_2$



## Lambda Calculus Syntax

- Or, more concisely, the syntax of a lambda calculus expression as context-free grammar is given by:

$$e = c \mid \text{id} \mid \lambda \text{id}.e \mid e_1\ e_2$$

- This is a production that defines the left hand side (here an expression $e$)
- Observe that this production is recursive
- With this production, we can now check if any expression is valid lambda calculus



## Lambda Calculus Syntax

- Consider the expression: $A = (\lambda x.x)\ 3$
- Now, recalling the syntax

$$e = c \mid \text{id} \mid \lambda \text{id}.e \mid e_1\ e_2$$

we can give a derivation proving that $A$ is valid

- $e \to e_1\ e_2 \to e_1\ 3 \to (\lambda x.e)\ 3 \to (\lambda x.x)\ 3$
- Any expression for which we can find a derivation is syntactically valid lambda calculus



## Are we done?

- We can now decide if any string is lambda calculus
- But we have no idea (yet) what these expressions mean!
- Just because we defined a syntax, this does not mean we have given meaning to expressions
- Giving meaning to syntax is called semantics
- Big chunk of this class: How to define syntax and semantics of programming languages

## Lambda calculus semantics

- Let's define the meaning for each expression in our production:
  - Constant $c$: The meaning of $c$ is the value of $c$
  - Identifier $id$: The meaning of $id$ is $id$
  - Lambda $\lambda x.e$: The meaning: $\lambda x.e$
  - Application $\lambda x.e\ e_2$: The meaning: $e[e_2/x]$
- $e[e_2/x]$ is substitution. We replace all free occurrences of $x$ by $e_2$ in expression $e$
- An occurrence of a variable is free if it is not bound by a $\lambda$
  Example: $(\lambda x.x)[2/x] = \lambda x.x$
- Upshot: We can define anonymous functions with binding operator $\lambda$.



## Examples

- Meaning (or value ) of $(\lambda x.x)\ 1$?
- $(\lambda x.x)\ 1 \to x[1/x] \to 1$
- $(\lambda x.(\lambda x.x)x)1 \to ((\lambda x.x)x)[1/x] \to (\lambda x.x)1 \to ...$
- Substitution is capture-avoiding: Does not replace variables bound by other $\lambda$'s
- Convention: We assume that $\lambda$-bindings extend as far to the right as possible
- We read $\lambda x.\lambda y.xy$ as $(\lambda x.(\lambda y.xy))$ But use parenthesis to be safe



## More Examples

- To make lambda calculus slightly more interesting, we will also allow arithmetic operators with their usual meaning.
- We could give them precise semantics, but too boring. We all know their semantics
- $(\lambda x.5 * x)\ 1 \to (5 * x)[1/x] \to (5 * 1) \to 5$
- $(\lambda x.\lambda y.x + y)\ 3\ 5 \to ((\lambda y.x + y)[3/x])\ 5 \to (\lambda y.3 + y)\ 5 \to (3 + y)[5/y] \to (3 + 5) \to 8$



## Properties of lambda expressions

- We have seen that to compute the value of lambda expressions, we only needed to define application: $\lambda x.e\ e_2$ as $e[e_2/x]$
- In lambda calculus, this is called $\beta$-reduction.
- Confluence: Order of reductions is provably irrelevant
- Other property of lambda expressions: $\lambda x.e \Leftrightarrow \lambda y.(e[y/x])$
- This is called $\alpha-$reduction
- Simply encodes that the name of lambda bound variables is irrelevant
- Analogy: $\int_0^\infty e^{-x}\,\mathrm{d}x \equiv \int_0^\infty e^{-y}\,\mathrm{d}y$



## Expression Equivalence

- Using $\alpha-$ and $\beta-$reductions, we can prove equivalence of expressions by computing their values using $\beta-$reduction and (if necessary) applying $\alpha-$reductions.
- Example: $e_1 = (\lambda x.x + 1)$ and $e_2 = (\lambda z.z + 1)$.
- Using $\alpha-$reduction, we can rewrite $e_1' = (\lambda x.x + 1) \to^{\alpha} (\lambda z.z + 1)$
- Have now proven that $e_1$ and $e_2$ are equivalent



## What else?

- Lambda calculus looks very far from a real programming language.
- On the face of it, many features missing.
  - Multi-argument functions
  - Declarations
  - Conditionals
  - Named Functions
  - Recursion
  - ...
- Next: How to express these features in basic lambda calculus

## Multi-argument functions

- How can we express adding two numbers?
- Recall earlier example: $(\lambda x.\lambda y.x + y)3\ 5$
- Here, we first reduce to $(\lambda x.\lambda y.x + y)\ 3\ 5 \to ((\lambda y.x + y)[3/x])\ 5 \to (\lambda y.3 + y)\ 5$
- In other words, we partially evaluate $\lambda x$, resulting in a new function $(\lambda y.3 + y)$.
- This is equivalent to having a $\lambda$-binding with multiple arguments
- This is known as Currying



## Declarations

- We want to be able to give names to subexpressions
- Equivalence in typical programming languages: Local declarations
- Specifically, we want to add a let-construct of the following form to lambda calculus
- let $x = e_1$ in $e_2$
- Insight: Can define meaning of let-construct in in terms of basic lambda calculus:



## Declarations

- Any ideas?
- One possibility: let $x = e_1$ in $e_2$ means $e_2[e_1/x]$
- Or equivalently: let $x = e_1$ in $e_2$ means $(\lambda x.e_2)e_1$
- Why are these definitions equivalent?



## Conditionals

- Conditional: if $x$ then $e_1$ else $e_2$
- Trick: We first define true and false as functions:
  let true $= (\lambda x \lambda y.x)$ let false $= (\lambda x \lambda y.y)$
- Recall: $\lambda$-bindings extend as far to the right as possible:
  $(\lambda x \lambda y.x) \equiv (\lambda x(\lambda y.x))$
- Then define conditional as:
  if $p$ then $e_1$ else $e_2 \to (\lambda p \lambda e_1 \lambda e_2.p\ e_1\ e_2)$
- Here,$p$ is a predicate, i.e. function evaluating to true or false
- Example predicates are EQZ, GTZ, etc.
- Observation: If we define numbers carefully in $\lambda$ calculus, we can also define those precisely, but we won't in class



## Named Functions

- We want to add functions with names
- Solution: Use the let-construct to name anonymous $\lambda$ terms:
- Write function definition as
  fun $f$ with $x = e_1$ in $e_2 \equiv$ let $f = (\lambda x.e_1)$ in $e_2$
- Function call is now just application $(f\ \ e_2) \to (\lambda x.e_1)e_2$



## Named Functions Examples

- How about a function that adds 3 to its argument?
- fun add with $x = x + 3$ in $e \to$ let add $= (\lambda x.x + 3)$ in $e$