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 ;-)
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.

- 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 partner 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

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 Syntax

- 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 $e_1 e_2$

Consider the expression: $A = (\lambda x.x) ~ 3$

Now, recalling the syntax

$$e = c \mid id \mid \lambda id.e \mid e_1 e_2$$

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

$$e \rightarrow e_1 e_2 \rightarrow e_1 3 \rightarrow (\lambda x.e) 3 \rightarrow (\lambda x.x) 3$$

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 $\lambda x.c$: The meaning: $\lambda x.c$
  - 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 $\lambda$.

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

- $(\lambda x.5 \times x) 1 \rightarrow (5 \times 1) \rightarrow 5$
- $(\lambda x.\lambda y.x + y) 3 \rightarrow ((\lambda y.x + y)[3/x]) 5 \rightarrow (\lambda y.3 + y) 5 \rightarrow (3 + y)[5/y] \rightarrow (3 + 5) \rightarrow 8$

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 $e_1 = (\lambda x.x + 1) \rightarrow^\alpha (\lambda z.z + 1)$

- Have now proven that $e_1$ and $e_2$ are equivalent

Examples

- Meaning (or value ) of $(\lambda x.x) 1$?

- $(\lambda x.x) 1 \rightarrow x[1/x] \rightarrow 1$

- $(\lambda x. (\lambda x.x)x) 1 \rightarrow ((\lambda x.x)[1/x]) (\lambda x.x) 1 \rightarrow ...$

- 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

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: $\int_0^\infty e^{-x} dx \equiv \int_0^\infty e^{-y} dy$

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
  \((\lambda x.\lambda y.x + y)\ 3\ \ 5 \to ((\lambda y.x + y)(\lambda y.3 + y))\ 5 \to (\lambda y.3 + y)\ 5 \)
- 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
  \[ \text{let } x = e_1 \text{ in } e_2 \]
- Or equivalently: let \( x = e_1 \) in \( e_2 \)
- Why are these definitions equivalent?

Conditionals

- Conditional: if \( x \) then \( e_1 \) else \( e_2 \)
- Trick: We first define true and false as functions:
  \[ \text{let } \text{true} = (\lambda x.1) \text{ and false} = (\lambda x.0) \]
- Recall: \( \lambda \)-bindings extend as far to the right as possible:
  \( (\lambda x.\lambda y.x) \equiv (\lambda x.\lambda y.y) \)
- Then define conditional as:
  \[ \text{if } p \text{ then } e_1 \text{ else } e_2 \to (\lambda p.\text{true} e_1 \text{ false} e_2) \]
- 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 
      \[ \text{fun } f \text{ with } x = e_1 \text{ in } e_2 \equiv \text{let } f = (\lambda x.e_1) \text{ in } e_2 \]
    - 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?
  - \[ \text{fun add with } x = x + 3 \text{ in } e \to \text{let add} = (\lambda x.x + 3) \text{ in } e \]