Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Normal forms
  - Reduction strategies
- Interpreters for the Lambda calculus
- Coding them in Haskell
Lambda Calculus

- A theory of functions
  - Theory behind functional programming
  - Turing-complete: any computable function can be expressed and evaluated using the calculus
  - “Lingua franca” of PL research

- Lambda ($\lambda$) calculus expresses function definition and function application
  - $f(x) = x \ast x$ becomes $\lambda x. x \ast x$
  - $g(x) = x + 1$ becomes $\lambda x. x + 1$
  - $f(5)$ becomes $\left(\lambda x. x \ast x\right) 5 \rightarrow 5 \ast 5 \rightarrow 25$
Syntax of Pure Lambda Calculus

- \( \lambda \)-calculus formulae (e.g., \( \lambda x. x y \)) are called expressions or terms

\[
E ::= x \mid (\lambda x. E_1) \mid (E_1 E_2)
\]

- A \( \lambda \)-expression is one of
  - Variable: \( x \)
  - Abstraction (i.e., function definition): \( \lambda x. E_1 \)
  - Application: \( E_1 E_2 \)
Syntactic Conventions

- Parentheses may be dropped from “stand-alone” terms \((E_1 E_2)\) and \((\lambda x. E)\)
  - E.g., \((f x)\) may be written as \(f x\)

- Function application groups from left-to-right (i.e., it is left-associative)
  - E.g., \(x y z\) abbreviates \((x y z)\)
  - E.g., \(E_1 E_2 E_3 E_4\) abbreviates \((((E_1 E_2) E_3) E_4)\)
  - Parentheses in \(x (y z)\) are necessary! Why?
Syntactic Conventions

- Application has higher precedence than abstraction
  - Another way to say this is that the scope of the dot extends as far to the right as possible
  - E.g., $\lambda x. x \ z = \lambda x. ( x \ z ) = ( \lambda x. ( x \ z ) ) = ( \lambda x. x \ z ) \neq ( ( \lambda x. x ) \ z )$

- **WARNING:** This is the most common syntactic convention (e.g., Pierce 2002). However some books give abstraction higher precedence; you might have seen that different convention
Semantics of Lambda Calculus

- An expression has as its meaning the value that results after evaluation is carried out

- Somewhat informally, evaluation is the process of reducing expressions
  
  E.g., \((\lambda x.\lambda y. x + y) \ 3 \ 2 \ \rightarrow \ (\lambda y. 3 + y) \ 2 \ \rightarrow \ 3 + 2 = 5\)

  (Note: this example is just an informal illustration. There is no \(+\) in the pure lambda calculus!)
Free and Bound Variables

- Abstraction \((\lambda x. E)\) is also referred as binding
- Variable \(x\) is said to be bound in \(\lambda x. E\)

- The set of free variables of \(E\) is the set of variables that appear unbound in \(E\)
- Defined by cases on \(E\)
  - \(\text{Var } x\): \(\text{free}(x) = \{x\}\)
  - \(\text{App } E_1 E_2\): \(\text{free}(E_1 E_2) = \text{free}(E_1) \cup \text{free}(E_2)\)
  - \(\text{Abs } \lambda x. E\): \(\text{free}(\lambda x. E) = \text{free}(E) - \{x\}\)
Free and Bound Variables

- A variable \( x \) is **bound** if it is in the scope of a lambda abstraction: as in \( \lambda x. E \)
- Variable is free otherwise

1. \( (\lambda x. x) \, y \)

2. \( (\lambda z. z \, z) \, (\lambda x. x) \)

3. \( \lambda x. \lambda y. \lambda z. \, x \, z \, (y \, (\lambda u. u)) \)
Free and Bound Variables

- We must take free and bound variables into account when reducing expressions.

E.g., \((\lambda x.\lambda y. x y) (y w)\)

- First, rename bound \(y\) in \(\lambda y. x y\) to \(z\): \(\lambda z. x z\)

\((\lambda x.\lambda y. x y) (y w) \Rightarrow (\lambda x.\lambda z. x z) (y w)\)

- Why do we need to rename bound \(y\)?

- Second, apply the reduction rule that substitutes \((y w)\) for \(x\) in the body \((\lambda z. x z)\)

\((\lambda z. x z) [(y w)/x] \Rightarrow (\lambda z. (y w) z) = \lambda z. y w z\)
Substitution, formally

- \((\lambda x. E) \, M \rightarrow E[M/x]\) replaces all free occurrences of \(x\) in \(E\) by \(M\)

- \(E[M/x]\) is defined by cases on \(E\):
  - \(\text{Var: } y[M/x] = \begin{cases} \text{M if } x = y \\ y \text{ otherwise} \end{cases}\)
  - \(\text{App: } (E_1 \, E_2)[M/x] = (E_1[M/x] \, E_2[M/x])\)
  - \(\text{Abs: } (\lambda y. E_1)[M/x] = \begin{cases} \lambda y. E_1 \text{ if } x = y \\ \lambda z.((E_1[z/y])[M/x]) \text{ otherwise,} \end{cases}\)

where \(z\) NOT in \(\text{free}(E_1) \cup \text{free}(M) \cup \{x\}\)
Substitution, formally

\[(\lambda x. \lambda y. x y) \ (y \ w)\]

\[\rightarrow (\lambda y. x y) [(y \ w)/x]\]

\[\rightarrow \lambda 1_. \ ( ((x \ y)[1_/y]))[(y \ w)/x] \)

\[\rightarrow \lambda 1_. \ ( (x \ 1_)[(y \ w)/x] \)

\[\rightarrow \lambda 1_. \ ( (y \ w) \ 1_ )\]

\[\rightarrow \lambda 1_. \ y \ w \ 1_\]

You will have to implement precisely this substitution algorithm in Haskell
Rules (Axioms) of Lambda Calculus

- **α rule (α-conversion):** renaming of parameter (choice of parameter name does not matter)
  - \( \lambda x. E \to_\alpha \lambda z. (E[z/x]) \) provided \( z \) is not free in \( E \)
  - e.g., \( \lambda x. x \ x \) is the same as \( \lambda z. z \ z \)

- **β rule (β-reduction):** function application (substitutes argument for parameter)
  - \( (\lambda x. E) \ M \to_\beta E[M/x] \)
  - Note: \( E[M/x] \) as defined on previous slide!
  - e.g., \( (\lambda x. x) \ z \to_\beta z \)
Rules of Lambda Calculus: Exercises

- Use $\alpha$-conversion and/or $\beta$-reduction:

  \[(\lambda x. x) \, y \rightarrow_{\alpha\beta} ?\]

  \[(\lambda x. x) \, (\lambda y. y) \rightarrow_{\alpha\beta} ?\]

  \[(\lambda x. \lambda y. \lambda z. x \, z \, (y \, z)) \, (\lambda u. u) \, (\lambda v. v) \rightarrow_{\alpha\beta} \]

  Notation: $\rightarrow_{\alpha\beta}$ denotes that expression on the left reduces to the expression on the right, through a sequence $\alpha$-conversions and $\beta$-reductions.
Rules of Lambda Calculus: Exercises

\[(\lambda x. \lambda y. \lambda z. x \ z \ (y \ z)) \ (\lambda u. \ u) \ (\lambda v. \ v) \ \rightarrow_{\alpha\beta}\]
Reductions

- An expression $(\lambda x. E) \ M$ is called a redex (for reducible expression)

- An expression is in normal form if it cannot be $\beta$-reduced

- The normal form is the meaning of the term, the “answer”
Definitions of Normal Form

- Normal form (NF): a term without redexes
- Head normal form (HNF)
  - $x$ is in HNF
  - $(\lambda x. E)$ is in HNF if $E$ is in HNF
  - $(x \ E_1 \ E_2 \ ... \ E_n)$ is in HNF
- Weak head normal form (WHNF)
  - $x$ is in WHNF
  - $(\lambda x. E)$ is in WHNF
  - $(x \ E_1 \ E_2 \ ... \ E_n)$ is in WHNF
Questions

- $\lambda z. z \ z$ is in NF, HNF, or WHNF?
- $(\lambda z. z \ z) \ (\lambda x. x)$ is in?
- $\lambda x. \lambda y. \lambda z. \ x \ z \ (y \ (\lambda u. u))$ is in?

- $(\lambda x. \lambda y. \ x) \ z \ ((\lambda x. \ z \ x) \ (\lambda x. \ z \ x))$ is in?
- $z \ ((\lambda x. \ z \ x) \ (\lambda x. \ z \ x))$ is in?
- $(\lambda z. (\lambda x. \lambda y. \ x) \ z \ ((\lambda x. \ z \ x) \ (\lambda x. \ z \ x)))$ is in?

Program Analysis CSCI 4450/6450, A Milanova
Simple Reduction Exercise

- $C = \lambda x.\lambda y.\lambda f. \ f \ x \ y$
- $H = \lambda f. \ f \ (\lambda x.\lambda y. \ x)$
- $T = \lambda f. \ f \ (\lambda x.\lambda y. \ y)$

What is $H \ (C \ a \ b)$?

$\rightarrow (\lambda f. \ f \ (\lambda x.\lambda y. \ x)) \ (C \ a \ b)$
$\rightarrow (C \ a \ b) \ (\lambda x.\lambda y. \ x)$
$\rightarrow ((\lambda x.\lambda y.\lambda f. \ f \ x \ y) \ a \ b) \ (\lambda x.\lambda y. \ x)$
$\rightarrow (\lambda f. \ f \ a \ b) \ (\lambda x.\lambda y. \ x)$
$\rightarrow (\lambda x.\lambda y. \ x) \ a \ b$

$\rightarrow a$
Exercise

- \( S = \lambda x. \lambda y. \lambda z. \ x \ z \ (y \ z) \)
- \( I = \lambda x. \ x \)
- What is \( S \ I \ I \ I ? \)

\[
( \lambda x. \lambda y. \lambda z. \ x \ z \ (y \ z) ) \ I \ I \ I \\
\rightarrow (\lambda y. \lambda z. \ I \ z \ (y \ z)) \ I \ I \\
\rightarrow (\lambda z. \ I \ z \ (I \ z)) \ I \\
\rightarrow I \ I \ (I \ I) = (\lambda x. \ x) \ I \ (I \ I) \\
\rightarrow I \ (I \ I) = (\lambda x. \ x) \ (I \ I) \\
\rightarrow I \ I = (\lambda x. \ x) \ I \rightarrow I
\]

An expression with no free variables is called **combinator**. \( S, I, C, H, T \) are combinators.

Reducible expression is underlined at each step.
Aside: Trace Semantics

- Models program execution
- In the imperative world
  - Basic operation: assignment statement
  - Execution (transition system) is a sequence of state transitions

Assignment: $\ell_j : x = E; \ell_i : \ldots$

$$(\ell_j, \sigma) \rightarrow (\ell_i, \sigma[x \leftarrow |E|](\sigma))$$

Assignment: $\ell_j : x = E_1 \text{ Op } E_2; \ell_i : \ldots$

$$(\ell_j, \sigma) \rightarrow (\ell_i, \sigma[x \leftarrow |E_1|](\sigma) \text{ Op } |E_2|)(\sigma))$$
Aside: Trace Semantics

- In the functional world
  - Basic operation is function application
  - Execution (transition system) is a sequence of \( \beta \)-reductions

\[
(\lambda x. \lambda y. \lambda z. \, x \, z \, (y \, z)) \rightarrow \ \\
(\lambda y. \lambda z. \, l \, z \, (y \, z)) \rightarrow \ \\
(\lambda z. \, l \, z \, (l \, z)) \rightarrow \ \\
\ldots \ \\
\lambda x. x
\]
Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Normal form
- Reduction strategies
- Lambda calculus interpreters
- Coding them in Haskell
Reduction Strategy

Let us look at \((\lambda x.\lambda y.\lambda z. x z (y z)) (\lambda u. u) (\lambda v. v)\)

Actually, there are (at least) two “reduction paths”:

Path 1:

\[
\begin{align*}
(\lambda x.\lambda y.\lambda z. x z (y z)) (\lambda u. u) (\lambda v. v) & \to_{\beta} \\
(\lambda y.\lambda z. (\lambda u. u) z (y z)) (\lambda v. v) & \to_{\beta} \\
(\lambda z. (\lambda u. u) z ((\lambda v. v) z)) & \to_{\beta} (\lambda z. z ((\lambda v. v) z)) \\
\lambda z. z z
\end{align*}
\]

Path 2:

\[
\begin{align*}
(\lambda x.\lambda y.\lambda z. x z (y z)) (\lambda u. u) (\lambda v. v) & \to_{\beta} \\
(\lambda y.\lambda z. (\lambda u. u) z (y z)) (\lambda v. v) & \to_{\beta} \\
(\lambda y.\lambda z. z (y z)) (\lambda v. v) & \to_{\beta} (\lambda z. z ((\lambda v. v) z)) \\
\lambda z. z z
\end{align*}
\]
Reduction Strategy

- A reduction strategy (also called evaluation order) is a strategy for choosing redexes
  - How do we arrive at the normal form (answer)?
- Applicative order reduction chooses the leftmost-innermost redex in an expression
  - Also referred to as call-by-value reduction
- Normal order reduction chooses the leftmost-outermost redex in an expression
  - Also referred to as call-by-name reduction
Reduction Strategy: Examples

- Evaluate \((\lambda x. \, x \, x) \, (\lambda y. \, y) \, (\lambda z. \, z)\) 
- Using applicative order reduction:
  \((\lambda x. \, x \, x) \, (\lambda y. \, y) \, (\lambda z. \, z)\) 
  \rightarrow (\lambda x. \, x \, x) \, (\lambda z. \, z)\) 
  \rightarrow (\lambda z. \, z) \, (\lambda z. \, z) \rightarrow (\lambda z. \, z)\) 
- Using normal order reduction:
  \((\lambda x. \, x \, x) \, (\lambda y. \, y) \, (\lambda z. \, z)\) 
  \rightarrow (\lambda y. \, y) \, (\lambda z. \, z) \, (\lambda y. \, y) \, (\lambda z. \, z)\) 
  \rightarrow (\lambda z. \, z) \, (\lambda y. \, y) \, (\lambda z. \, z)\) 
  \rightarrow (\lambda y. \, y) \, (\lambda z. \, z) \rightarrow (\lambda z. \, z)\)
Reduction Strategy

- In our examples, both strategies produced the same result. This is not always the case.
  - First, look at expression \((\lambda x. x x) (\lambda x. x x)\). What happens when we apply \(\beta\)-reduction to this expression?
  - Then look at \((\lambda z. y) ((\lambda x. x x) (\lambda x. x x))\)
    - Applicative order reduction – what happens?
    - Normal order reduction – what happens?
Church-Rosser Theorem

- Normal form implies that there are no more reductions possible

Church-Rosser Theorem, informally

- If normal form exists, then it is unique (i.e., result of computation does not depend on the order that reductions are applied; i.e., no expression can have two distinct normal forms)
- If normal form exists, then normal order will find it
Reduction Strategy

- Intuitively:

- Applicative order (call-by-value) is an eager evaluation strategy. Also known as strict

- Normal order (call-by-name) is a lazy evaluation strategy

- What order of evaluation do most PLs use?
Exercises

- Evaluate \((\lambda x.\lambda y. x y) ((\lambda z. z) w)\)
- Using applicative order reduction

- Using normal order reduction
An interpreter for the lambda calculus is a program that reduces lambda expressions to "answers".

We must specify:
- The definition of "answer". Which normal form?
- The reduction strategy. How do we choose redexes in an expression?
An Interpreter

Definition by cases on \( E ::= x \mid \lambda x. E_1 \mid E_1 E_2 \)

interpret(\( x \)) = \( x \)

interpret(\( \lambda x. E_1 \)) = \( \lambda x. E_1 \)

interpret(\( E_1 E_2 \)) = \begin{align*}
& \text{let } f = \text{interpret}(E_1) \\
& \text{in case } f \text{ of} \\
& \quad \lambda x. E_3 \rightarrow \text{interpret}(E_3[E_2/x]) \\
& \quad - \rightarrow f E_2
\end{align*}

- What normal form: Weak head normal form
- What strategy: Normal order

Haskell syntax:
let .... in
case f of
-
Another Interpreter

- Definition by cases on $E ::= x \mid \lambda x. E_1 \mid E_1 E_2$

  - $\text{interpret}(x) = x$
  - $\text{interpret}(\lambda x. E_1) = \lambda x. E_1$
  - $\text{interpret}(E_1 E_2) = \text{let } f = \text{interpret}(E_1)\text{ in case } f \text{ of }$
    - $\lambda x. E_3 \rightarrow \text{interpret}(E_3[a/x])$
    - $\rightarrow f \; a$

- What normal form: Weak head normal form
- What strategy: Applicative order
Outline

- Pure lambda calculus, a review
  - Syntax and semantics
  - Free and bound variables
  - Rules (alpha rule, beta rule)
  - Reduction strategies
  - Normal form
- Lambda calculus interpreters
- Coding them in Haskell
Coding them in Haskell

- In HW5 you will code an interpreter in Haskell
- Haskell
  - A functional programming language

Key ideas
- Lazy evaluation
- Static typing and polymorphic type inference
- Algebraic data types and pattern matching
- Monads … and more
Lazy Evaluation

- Unlike Scheme (and most programming languages), Haskell does **lazy evaluation**, i.e., **normal order reduction**
  - It won’t evaluate an argument `expr` until it is needed

```haskell
> f x = [] // f takes x and returns the empty list
> f (repeat 1) // returns?
> []
> head (tail [1..]) // returns?
> 2 // [1..] is infinite list of integers
```

- Lazy evaluation allows us to work with infinite structures!
Unlike Scheme, which is dynamically typed, Haskell is `statically typed`!

Unlike Java/C++ we don’t always have to write type annotations. Haskell `infers` types!
- A lot more on type inference later!

\[ f \ x = \text{head} \ x \] // \( f \) returns the head of list \( x \)

\[ f \ True \] // returns?
- Couldn't match expected type `[a]` with actual type `Bool`
- In the first argument of `f`, namely `True`
  In the expression: `f True` …
Algebraic Data Types

- Algebraic data types are **tagged unions** (aka sums) of **products** (aka records)

```
data Shape = Line Point Point | Triangle Point Point Point | Quad Point Point Point Point
```

- Haskell keyword
- **new constructors** (a.k.a. **tags**, **disjuncts**, **summands**)
- Line is a binary constructor, Triangle is a ternary ...
Algebraic Data Types in HW5

- Constructors create new values
- Defining a lambda expression
  
  ```
  type Name = String
  data Expr = Var Name
  | Lambda Name Expr
  | App Expr Expr
  ```

  ```>
  e1 = Var "x" // Lambda term \( \lambda x.x \)
  > e2 = Lambda "x" e1 // Lambda term \( \lambda x.x \)
  ```
Examples of Algebraic Data Types

data Bool = True | False

data Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun

data List a = Nil | Cons a (List a)

data Tree a = Leaf a | Node (Tree a) (Tree a)

data Maybe a = Nothing | Just a

Maybe type denotes that result of computation can be a or Nothing. Maybe is a monad.
Data Constructors vs Type Constructors
Pattern Matching

- Examine values of an algebraic data type

```haskell
anchorPnt :: Shape \rightarrow Pnt
anchorPnt s = case s of
  Line     p1 p2 \rightarrow p1
  Triangle p3 p4 p5 \rightarrow p3
  Quad     p6 p7 p8 p9 \rightarrow p6
```

- Two points
  - Test: does the given value match this pattern?
  - Binding: if value matches, bind corresponding values of \( s \) and pattern

Type signature of anchorPnt: takes a Shape and returns a Pnt.
isFree :: Name → Expr → Bool

isFree v e =
  case e of
    Var n → if (n == v) then True else False
    Lambda ...
Haskell Resources

- http://www.seas.upenn.edu/~cis194/spring13/
- https://www.haskell.org/