Quick announcement

Midterm date is Wednesday Oct 24, 11-12pm.
The lambda calculus

\[
<\text{expr}> = \text{ID} \quad \downarrow \\
\mid (\lambda \text{ID} . \ <\text{expr}> ) \\
\mid ( <\text{expr}> \ <\text{expr}> )
\]

- parameter
- body
- call

- func
- arg
The lambda calculus (Racket)

<expr> = ID
     | (lambda (ID) <expr>)
     | (<expr> <expr>)
The lambda calculus (Haskell)

<expr> = ID
    | \ ID -> <expr>
    | <expr> <expr>
The lambda calculus (Python)

\[<\text{expr}> = \text{ID} \]
\[\quad | \quad \text{lambda } \text{ID} : <\text{expr}>\]
\[\quad | \quad <\text{expr}> ( <\text{expr}> )\]
Functional programming

a programming paradigm centred on evaluating functions
Question: What is a program?
Simplest answer: a single expression.

<prog> = <expr>
<expr> = ...
What does it mean to “run” such a program? To *evaluate* the expression.

\[ \langle \text{prog} \rangle = \langle \text{expr} \rangle \\
\langle \text{expr} \rangle = \ldots \]
Semantics

the meaning of the elements of a language
Denotational semantics

the abstract mathematical value of an expression

[intuitively, based on our knowledge of abstract domains, e.g. arithmetic]
Operational semantics

the rules that govern how an expression is evaluated

[based on our model of how computation occurs]
10

3 + 7

(*) 2 5

(\x \rightarrow x + 3) 7

(first (list 10 20 30 40))

ord('n')

head (tail [9, 10, 11, 12])
In the lambda calculus, the denotational semantics use just one idea: function calls are evaluated using substitution.

$$(((\lambda\ x\ .\ \ x)\ y)\ ==\ y$$
In our limited set of Racket/Haskell, the denotational semantics use two ideas: function calls as substitution, and known operations on primitive data types.

\[
   \text{((lambda (x) (+ x 10)) 20) =\Rightarrow (+ 20 10) =\Rightarrow 30}
\]
The operational semantics may seem straightforward (“call stack”)... more on this throughout the course.
In the lambda calculus, the denotational semantics use just one idea: function calls are evaluated using substitution.

The only thing a function can do is return a value. No mutation, no I/O.
Functional programming

a programming paradigm centred on evaluating mathematical (or pure) functions

(as a consequence, values are immutable)
Name binding

an association of an identifier to an expression
In Racket:
<binding> =
  (define ID <expr>)

In Haskell:
<binding> =
  ID = <expr>
a program is an expression to be evaluated, but we can include name bindings for readability

<prog> = <binding> ... <expr>

<binding> = ...
<expr> = ...
In our limited set of Racket/Haskell, the denotational semantics use four ideas:

- function calls as substitution
- known operations on primitive data types
- name bindings (definitions)
- name lookup
in pure functional programming, bindings are fixed, and cannot be reassigned

names are **referentially transparent**: they can be replaced by their corresponding value everywhere in the program without changing the program’s meaning
(define nums
  (list 1 2 3))

nums = [1, 2, 3]

; lots of code
(f nums)

# lots of code
f(nums)

; lots of code
(g nums)

# lots of code
g(nums)
(define nums
  (list 1 2 3))

; lots of code
(f (list 1 2 3))

; lots of code
(g (list 1 2 3))
unbounded data

a review of structural recursion
A natural number is:
- 0
- 1 + n, where n is a nat.

A list is:
- empty
- x “+” L, where x is a value and L is a list.

\[
\text{cons} \quad x = \text{[]} \quad L = [2, 3, 4] \\
\implies x + L = [2, 3, 4]
\]
structure of data -> structure of code
A generic template

\[
\begin{align*}
\text{(define (f lst)} & \\
\text{  (if (empty? lst)} & \\
\text{    \ldots} & \\
\text{    (\ldots (first lst)} & \\
\text{    \ldots)} & \\
\text{    \ldots)} & \\
\text{  \ldots)} & \\
\text{  \ldots (f (rest lst))} & \\
\text{  \ldots) )} & \\
\end{align*}
\]

\[
\begin{align*}
f \text{ lst} = \\
\text{  if null lst} & \\
\text{    \ldots} & \\
\text{  else \ldots (head lst)} & \\
\text{    \ldots} & \\
\text{  \ldots (f (rest lst))} & \\
\text{    \ldots)} & \\
\end{align*}
\]
Pattern-matching: value-based matching
\[
f(x) = \\
\begin{align*}
&\text{if } x == 0 \\
&\quad \text{then} \\
&\quad \quad 10 \\
&\text{else if } x == 1 \\
&\quad \text{then} \\
&\quad \quad 20 \\
&\text{else} \\
&\quad x + 30
\end{align*}
\]

\[
\begin{align*}
f(0) &= 10 \\
f(1) &= 20 \\
f(x) &= x + 30
\end{align*}
\]
Pattern-matching: structural matching
2: \[
g \text{lst} =
\begin{align*}
\text{if } \text{null lst} & \text{ then } 10 \\
\text{else} & \\
\text{let } x = \text{head lst} \\
xs = \text{tail lst} \\
in \\
x + \text{length xs}
\end{align*}
\]

\[
g [] = 10 \\
g (x:xs) = x + \text{length xs}
\]
Recursion, efficiency, and the difference between interface and implementation
(define (f n)
  (if (zero? n)
      0
      (+ 1 (f (- n 1))))
))

\[ f(100) \]
(define (f n)
  (if (zero? n)
      0
      (f (- n 1)))))
Let $E$ be an expression, and $E'$ be a subexpression in $E$.

$E'$ is in a **tail position** with respect to $E$ if evaluating $E'$ is the last step in evaluating $E$.

If $E'$ is a function call in tail position, it is a **tail call**.
Tail call elimination

An optimization that removes (i.e., deallocates) the current stack frame when a tail call is made.
Tail recursion

A recursive function is tail recursive when all recursive calls are tail calls.