The lambda calculus

\[ \text{<expr>} ::= \text{ID} \]
\[ \quad \mid (\lambda \text{ID} \ . \ \text{<expr>}) \]
\[ \quad \mid (\text{<expr>} \ \text{<expr>}) \]
The lambda calculus (Racket)

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

\[ <expr> ::= ID \mid \text{ID} \rightarrow <expr> \mid <expr> <expr> \]
The lambda calculus (Python)

<expr> ::= ID
    | lambda ID : <expr>
    | <expr> ( <expr> )
functional programming: a programming paradigm centred on evaluating functions
a program is an expression to be evaluated

\(<\text{prog}\> := \ <\text{expr}\>
\<\text{expr}\> := \ldots
Name bindings

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

\[
\text{prog} := [\text{binding} \ldots] \text{expr}
\]

\[
\text{expr} := \ldots
\]
in pure functional programming, bindings are *immutable*
functional programming: a programming paradigm centred on evaluating mathematical functions
(define nums
  (list 1 2 3))

; lots of code
(f nums)

; lots of code
(g nums)
(define nums
  (list 1 2 3))
nums = [1, 2, 3]

; lots of code
(f (list 1 2 3))
# lots of code
f([1, 2, 3])

; lots of code
(g (list 1 2 3))
# lots of code
g([1, 2, 3])
Semantics and evaluation
semantics: the meaning of the elements of a language
denotational semantics
abstract, mathematical meaning
10

3 + 7

(* 2 5)

(\x -> x + 3) 7

(first (list 10 20 30 40))

ord('\\n')

head (tail [9, 10, 11, 12])
operational semantics
how an expression is evaluated
The lambda calculus

\[ <\text{expr}> \ ::= \text{ID} \]
\[ | \ (\lambda \text{ID} \ . \ <\text{expr}>\) \]
\[ | \ (\text{<expr>} \ <\text{expr}>) \]
function call evaluation

substitution
Evaluation order
When all subexpressions are “valid”, evaluation order doesn’t matter.
strict denotational semantics
strict denotational semantics for function calls

if an argument expression is undefined, the call expression is undefined
typically implemented using
*left-to-right eager evaluation*
non-strict denotational semantics for function calls

if an argument expression is undefined, the entire is undefined only if that argument is required to evaluate the program*
def f(n):
    s = 0
    for _ in range(n):
        s += 1
    return s

f(100000)
(define f
  (lambda (x) 1))
(f (error "failed"))

f = \x \rightarrow 1
f (error "failed")
non-strict denotational semantics for function calls

if an argument expression is undefined, the entire is undefined only if that argument is required to evaluate the program*
In strict languages, functions can’t “skip” arguments.
(define (avg numbers)
  (if (equal? 0 (length numbers))
    0
    (/ (sum numbers) (length numbers)))))
short-circuiting
control flow structures
unbounded data: 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.
structure of data -> structure of code
call stack woes
tail position, tail recursion