Quick announcement

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

The lambda calculus

```
<expr> = ID
       | (λ ID . <expr>)
       | (<expr> <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)

```
<expr> = ID
       | lambda ID : <expr>
       | <expr> ( <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.

```
<prog> = <expr>
<expr> = ...
```

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 -> 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.

```
((λ 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.

```
    ((lambda (x) (+ x 10)) 20)
==> (+ 20 10)
==> 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))

; lots of code
(f nums)

; lots of code
(g nums)
```

```
nums = [1, 2, 3]

# lots of code
f(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))
```

```
nums = [1, 2, 3]

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

# lots of code
g([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.

x = 1 L = [2,3,4]

x :: L = [1,2,3,4]

cons

x = [ ] L = [2,3,4]

x :: L = [[ ],2,3,4]

structure of data -> structure of code

A generic template

```
(define (f lst)
  (if (empty? lst)
      ...
      (... (first lst)
           ...
           (f (rest lst))
           ...)))
```

```
f lst =
  if null lst
  then ...
  else ... (head lst)
           ...
           (f (rest lst))
           ...
```

Pattern-matching: value-based matching

```
f x =
    if x == 0
    then
        10
    else if x == 1
    then
        20
    else
        x + 30
```

```
f 0 = 10
f 1 = 20
f x = x + 30
```

Pattern-matching: structural matching

2 : []

```
g lst =
  if null lst
  then
    10
  else
    let x = head lst
        xs = tail lst
    in
      x + length xs
```

```
g [] = 10
g (x:xs) =
    x + length xs
```

g [1,2,3]

x = 1

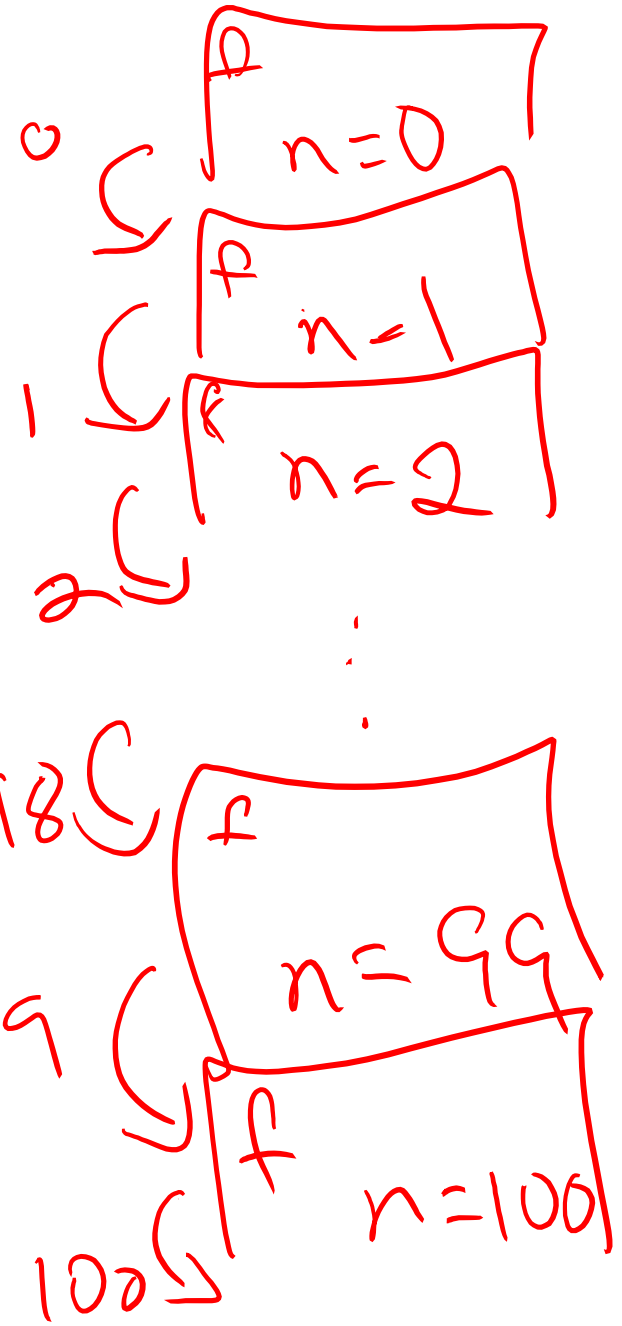
xs = [2,3]

Recursion, efficiency, and the difference between interface and implementation

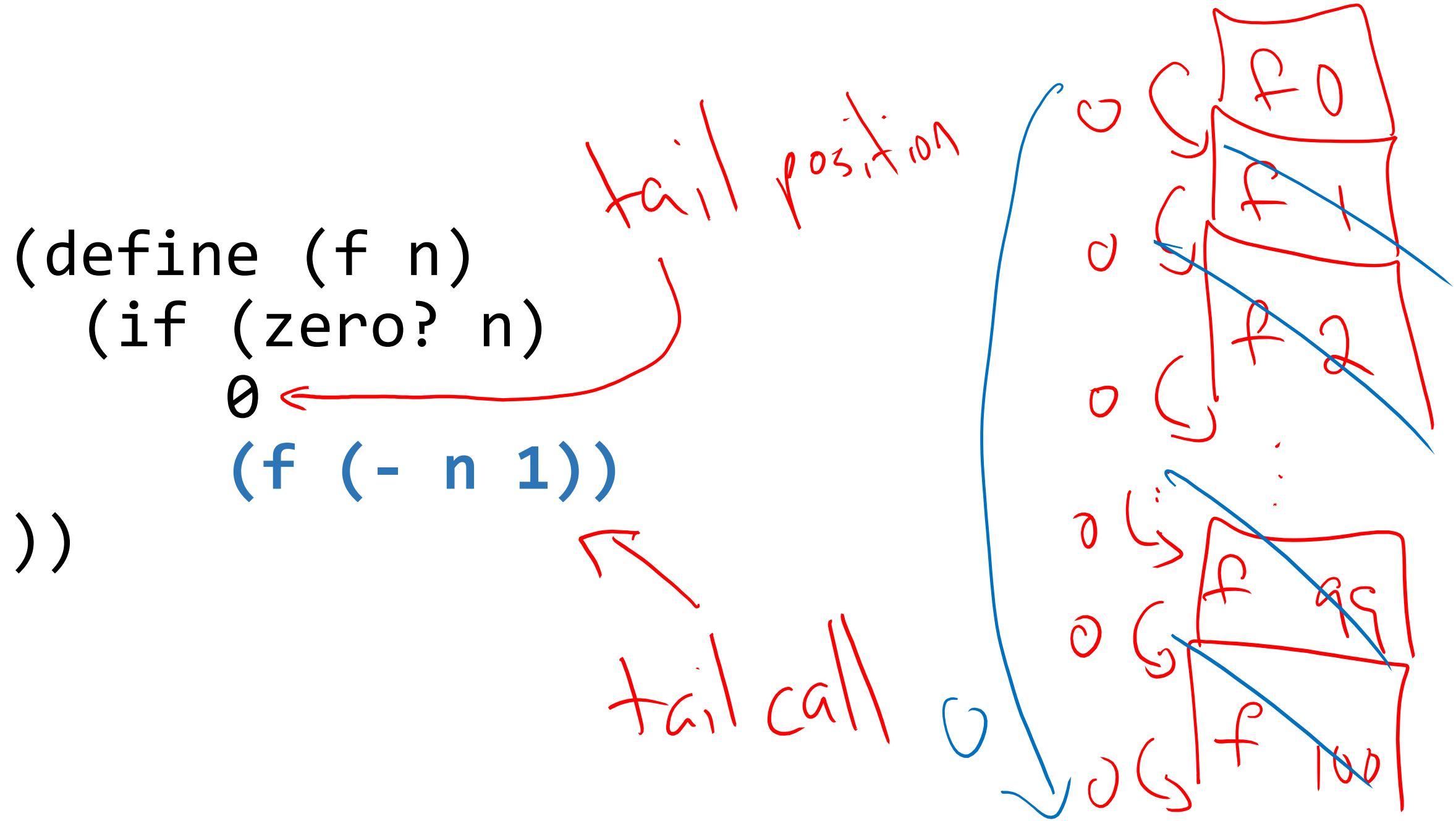
f(100)

```
(define (f n)
  (if (zero? n)
      0
      (+ 1 (f (- n 1)))
))
```

tail position (call)



```
(define (f n)
  (if (zero? n)
      0
      (f (- n 1))
))
```

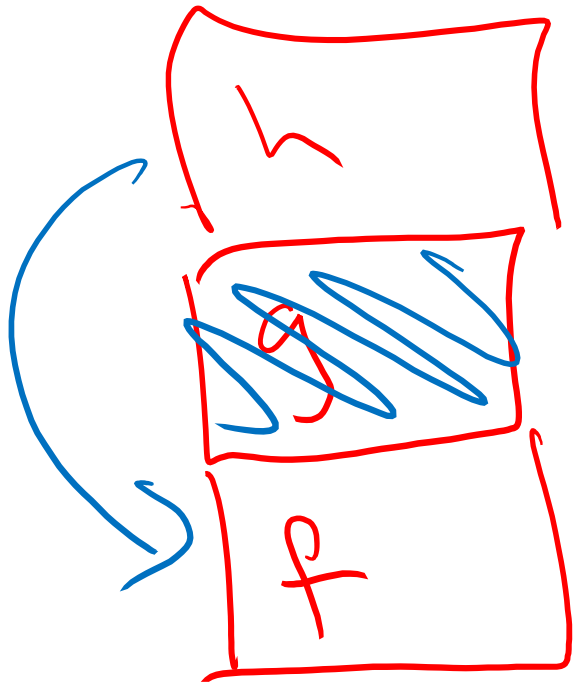


Let *E* be an expression, and *E'* be a subexpression in E.

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.

h is a tail call
in g

Tail recursion

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