Recursion

CS135 Lecture 04

L04.0 One weird trick

Sum of the natural numbers up to *n*

$$0 + 1 + 2 + 3 + 4 = 10$$
$$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 = 36$$
$$0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 = 91$$

Sum of the natural numbers up to *n*

From your math courses you might know:

$$0 + 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

But what does the "..." mean? What is it telling us to do?

Informally, the "..." describes a **computation**. It says, "start at zero and add increasing natural numbers until you reach *n*.

The result of this **computation** is the sum of the natural numbers up to *n*.

Sum of the natural numbers up to *n*

Let's try to describe this computation without resorting to informal notation ("...").

Let *f*(*n*) be the sum of the natural numbers up to *n*.

$$f(n) = 0 + 1 + 2 + 3 + ... + (n-1) + n$$

If we look closely, we can see this equation is made of two parts:

1. the sum of the natural numbers up to *n* - 1,
2. plus *n*.

$$f(n) = f(n-1) + n = n + f(n-1)$$

Sum of the natural numbers up to *n*

If we include the case that *f*(0) = 0, then we get:

$$f(n) = \begin{cases} 0, & \text{if } n = 0, \\ n + f(n-1), & \text{if } n > 0. \end{cases}$$

We have defined *f*(*n*) in terms of itself.

This function more precisely defines the computation that was implicit in the “...”.

Sum of the natural numbers up to *n*

$$
\begin{aligned}
f(6) &= 6 + f(5) \\
&= 6 + 5 + f(4) \\
&= 6 + 5 + 4 + f(3) \\
&= 6 + 5 + 4 + 3 + f(2) \\
&= 6 + 5 + 4 + 3 + 2 + f(1) \\
&= 6 + 5 + 4 + 3 + 2 + 1 + f(0) \\
&= 6 + 5 + 4 + 3 + 2 + 1 + 0
\end{aligned}
$$

Sum of the natural numbers up to *n*

We can convert this equation directly into Racket:

```
;; Sum the natural numbers up to n.
;; sum-to: Nat -> Nat
(define (sum-to n)
  (cond [(zero? n) 0]
        [else (+ n (sum-to (sub1 n)))]))
(check-expect (sum-to 4) 10)
(check-expect (sum-to 8) 36)
(check-expect (sum-to 13) 91)
(check-expect (sum-to 0) 0)
```

Recursion

A function defined in terms of itself in this way is called *recursive*. Typically, the arguments of a recursive call get “smaller” in some way until it reaches a “base case”. In the case of **`sum-to`**, the argument gets smaller by one, until it reaches the base case of zero.

The predicate **`zero?`** is **`true`** if its argument is `0`; **`false`** otherwise.

The function **`sub1: Num -> Num`** subtracts `1` from its argument.

In the rest of this lecture we will use **`sub1`** to make arguments smaller and **`zero?`** to test the base case.

One weird trick

It is a central result in the theory of computation that (informally stated) anything a computer can compute we can now compute, since we have:

1. arbitrarily large numbers,
2. conditional expressions, and
3. recursion.

Our model of computation is “Turing complete”

Very informally, a model of computation is **Turing complete** if it can compute anything that any other computer can, given enough time and memory.

All the phone, laptops, and other computers in the room are Turing complete. Our model of computation, with its carefully defined substitution steps that can be carried out with a pencil and paper, is just as powerful. But much slower.

A Turing complete model of computation is as expressive as any other—it can carry out any computation that any other computer can, given enough time and memory.

Alan Turing (1912-1954)

In a 1936 paper, British mathematician Alan Turing showed that a simple abstract machine—now called a **Turing machine**—can perform any computation that any other programmable computer can, laying the foundation for the concept of Turing completeness.

We will learn more about Alan Turing, and his role in the foundation of the field of Computer Science in the history lecture at the end of the term.

L04.1 The Rules for Recursion

Recursion can be hard to get right

For now, we will keep recursion as simple as possible. We will always follow the same basic *template*:

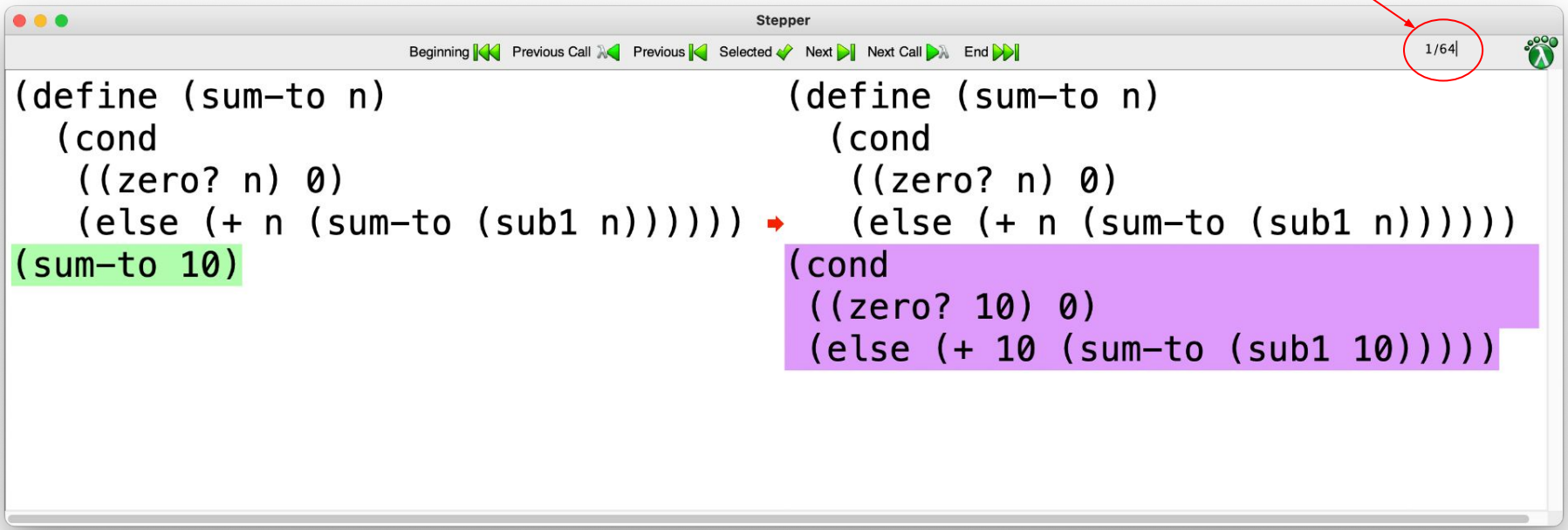
```
(define (natural-template n)
  (cond
    [(zero? n) ...]
    [... ...]
    [else (... n ... (natural-template (sub1 n)))]))
```

In this case the ellipses ("...") just mean that we can add problem-specific code at those positions.

Three properties of functions we consider in CS135

1. **Termination.** Our template guarantees termination. Since the argument gets smaller by one, it eventually reaches the base case of zero.
2. **Correctness.** We could prove the correctness of `sum-to` by induction. If you've seen proof by induction in a math class, you may be able to construct a proof yourself. In this class we use testing to help ensure correctness.
3. **Efficiency.** In CS135 we measure efficiency by the number of substitution steps, as measured by the stepper.

The number of substitution steps, as measured by the stepper



Rules for recursion (first version)

1. Change one argument closer to termination while recurring. No other arguments can change.
2. When recurring on a natural number use `(sub1 n)` and test termination with `zero?`

In this version, the argument in #1 is always the same argument.

This is the first version of our rules. Will be develop a final version over the coming lectures.

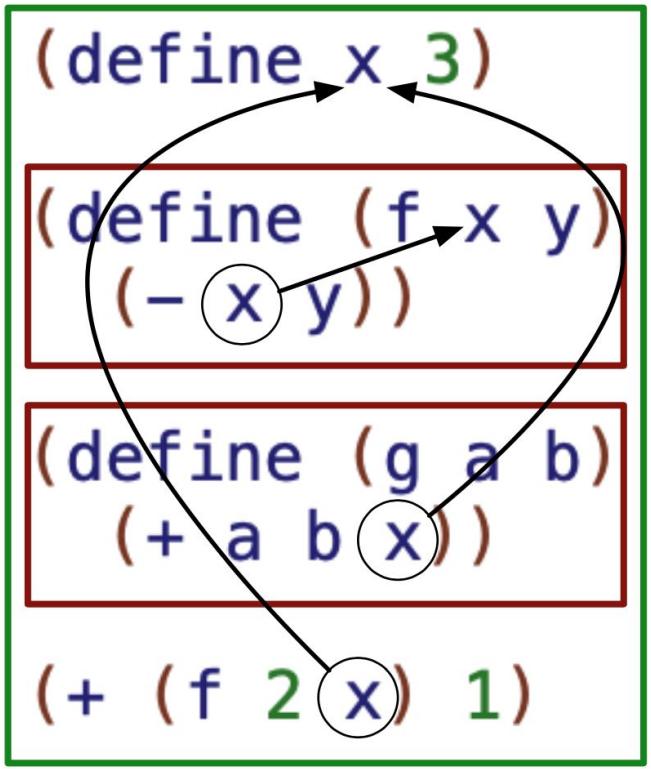
L04.2 Names and scope

Scope

Now that we have functions defined in terms of themselves, it's worth thinking a little more about the names used for functions, parameters and contents (collectively called *identifiers*) especially when the same identifier is used multiple times.

The **scope** of an identifier is where it has effect within the program.

Scope



Two kinds of scope (for now):
global and *function*

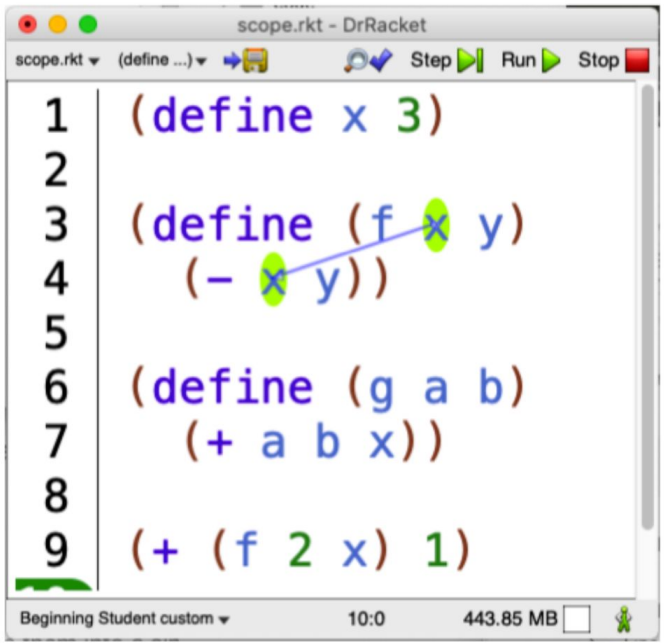
The smallest enclosing scope has priority

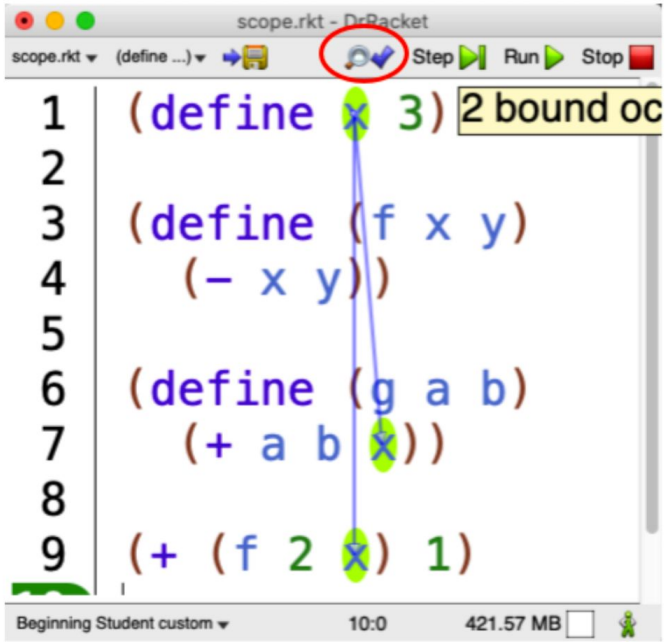
Duplicate identifiers within the same scope will cause an error:

```
(define f 3)
(define (f x) (sqr x))
Racket Error: f: this name was…
```

Scoping tools in DrRacket

DrRacket can help you determine an identifier's scope.





L04.3 Our design recipe

Example: The product of the natural numbers to *n*

$$1 \times 2 \times 3 \times ... \times n - 1 \times n = n!$$

$$1 \times 2 \times 3 = 3! = 6$$
$$1 \times 2 \times 3 \times 4 \times 5 = 5! = 120$$
$$1 \times 2 \times 3 \times 4 \times 5 \times 6 \times 7 = 7! = 5040$$

Called “*n* factorial” and written “*n*!”. The value grows quickly as *n* increases.

There is no simple formula for *n*!. We just have to compute it.

We start multiplying from 1. If we started from 0, *n*! would be easy to compute.

To make things simple, we assume 0! = 1, which is a widely accepted definition.

Start by writing a purpose, contract and header

From the mathematical definition we that the factorial function will consume a **`Nat`** and produce a **`Nat`**. An obvious name is **`factorial`** but **`!`** would also be fine.

```
;; Compute n!
;; factorial: Nat -> Nat
(define (factorial n) ...)
```

The **purpose** should indicate what the function produces and consumes, unless it is clear from the contract.

The **contract** tells us how the function is used.

The **heade**r gives the name of the function and name(s) for the parameter(s).

The **body** of the function will replace the ellipses (“...”)..

Write example test cases to aid understanding

```
;; Compute n!
;; factorial: Nat -> Nat
(define (factorial n) ...)

(check-expect (factorial 3) 6)
(check-expect (factorial 5) 120)
(check-expect (factorial 7) 5040)
(check-expect (factorial 0) 1)
(check-expect (factorial 1) 1)
```

The last two examples are reasonable boundary cases. We don't need to worry about negative numbers or inexact numbers because the function consumes **`Nat`**.

Mathematical definition of *f*(*n*) = *n*!

To define the body, let's look at the mathematical definition of *n*!

$$f(n) = \begin{cases} 1, \text{if } n = 0, \\ n \times f(n-1), \text{if } n > 0. \end{cases}$$

Think about *f*(1).

Think about *f*(3).

How would this function translate to Racket?

Starting with a template, write the body of the function

To write the function body, we can start with the template and fill in names from the header.

```
(define (factorial n)
  (cond
    [(zero? n) ...]
    [... ...]
    [else (... n ... (factorial (sub1 n)))]))
```

Starting with a template, write the body of the function

We can write the rest of the body from the mathematical definition.

```
(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))
```

Putting it all together

```
;; Compute n!
;; factorial: Nat -> Nat
(define (factorial n)
  (cond
    [(zero? n) 1]
    [else (* n (factorial (sub1 n)))]))
(check-expect (factorial 3) 6)
(check-expect (factorial 5) 120)
(check-expect (factorial 7) 5040)
(check-expect (factorial 0) 1)
(check-expect (factorial 1) 1)
```

Write additional tests, if needed

After putting the pieces together, you might discover a boundary case you missed, or that otherwise your examples don’t provide full coverage.

As a last step, add additional test cases.

For `factorial` our initial tests provide full coverage.

Our design recipe

We follow this “design recipe” when solving problems in CS135.

1. Write down the **purpose** of the function. Understand the problem.
2. Write a **contract** and **header** for the function.
3. Write **examples** to aid understanding (work them out by hand).
4. Starting with a template, write the **body** of the function.
5. Write additional **tests**, if needed, especially to catch boundary cases.

Initially, we will be explicit in following these steps. Later, you can do many of the steps in your head, but you can always fall back to this explicit design recipe if you are having trouble with a problem.

A simpler name?

```
;; Compute n factorial (n!)
;; !: Nat -> Nat
(define (! n)
  (cond
    [(zero? n) 1]
    [else (* n (! (sub1 n)))]))
(check-expect (! 3) 6)
(check-expect (! 5) 120)
(check-expect (! 7) 5040)
(check-expect (! 0) 1)
(check-expect (! 1) 1)
```

This name is simpler, but is it better?

Lecture 04 Summary

What happens next?

Over four lectures we ~~will~~ develop**ed** our model of computation:

1. Values and expressions
2. Functions
3. Conditional expressions
4. **Recursion**

After the final step, we ~~will~~ have built a complete “computer”, essentially from math.

We will ~~then~~ now add “lists” to our model of computation to simplify data organization.

We will then explore a variety of basic algorithms and data structures using lists.

L04: You should know

- How to use our design recipe to write functions
- How to write recursive functions following version 1 of the Rules for Recursion
- How to use **`zero?`** and **`sub1`** to write recursive functions
- How the Rules for Recursion (version 1) guarantee termination
- How to use the stepper to measure efficiency.

L04: Allowed constructs

Newly allowed constructs:

`sub1`

Recursion (following the first version of the rules)

Previously allowed constructs:

```
( ) [ ] + - * / = < > <= >= ;
abs acos and asin atan boolean? check-expect check-within
cond cos define e else exp expt false inexact? integer? log
max min not number? or pi quotient rational? remainder sin
sqr sqrt symbol? symbol=? tan true zero?
anyof Bool Int Nat Num Rat Sym
```