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*.

So, the result of this **computation** is *n*(*n* + 1)/2.

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-to consumes a natural number n and produces the sum
;; of 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*. Usually 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.

L04.01 The Rules of 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*:

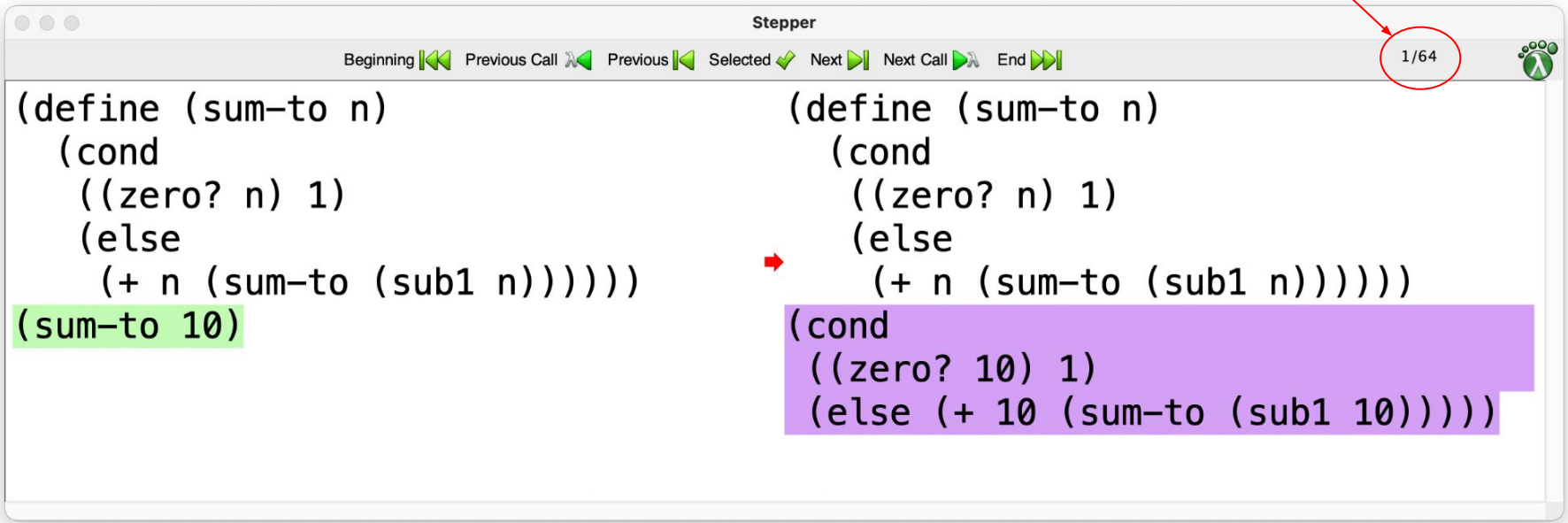
```
(define (natural-template n)
  (cond
    [(zero? n) ...]
    [... ...]
    [else (... n (natural-template (sub1 n)))]))
```

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 used by the stepper



Rules of 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?`

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

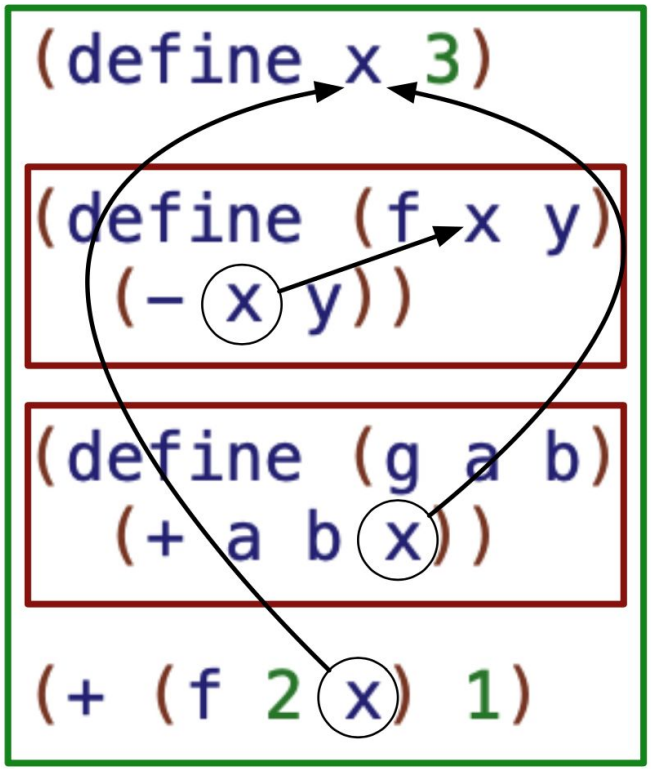
L04.02 Names and scope

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*

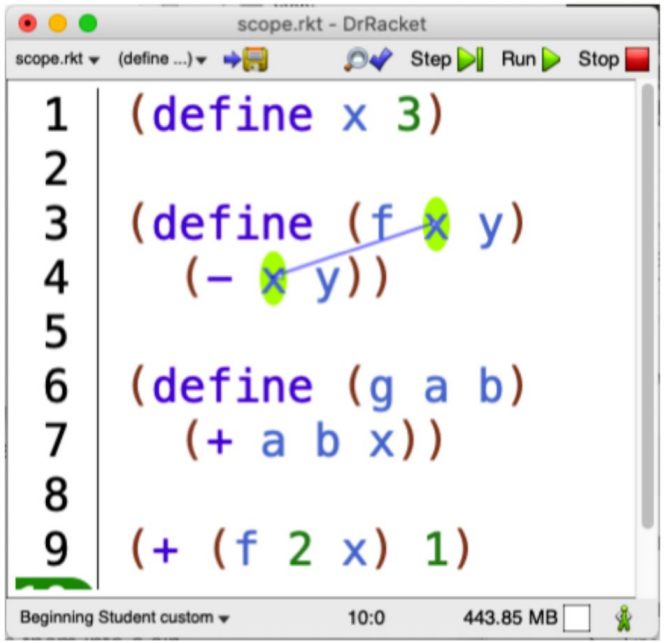
The smallest enclosing scope has priority

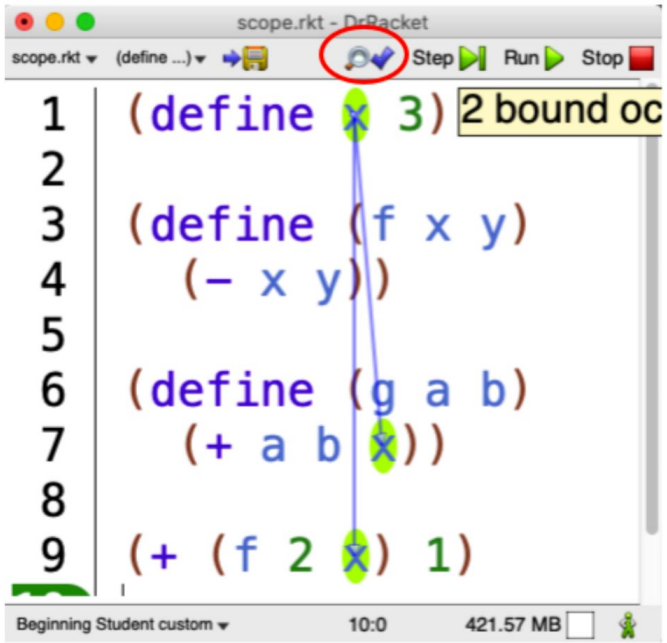
Duplicate identifiers within the same scope will cause an error:

```
(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.





L04.03 Factorial!

Product of the natural numbers to *n* (except 0)

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

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

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.

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

Computing *n*!

Purpose:

```
;; factorial consumes a natural number n and produces n!.
```

Examples:

```
(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.

Function header and contract

The factorial function will take a Nat and produce a Nat, so we can now write at least the “header” of our function and a contract.

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

The header gives the name of the function and name(s) for the parameter(s).

The “body” of the function replaces the ellipses (“...”) above.

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

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

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

Think about $f(1)$.

Writing the function body

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 (factorial (sub1 n))]))
```

Writing the function body

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

```
;; factorial consumes a natural number n and produces 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)
```

Design pattern

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

1. Write a draft of the **purpose**. Understand the problem.
2. Write **examples** to aid understanding (by hand, then using `check-expect`).
3. Write a function definition **header** and **contract**.
4. Finalize the **purpose**, adding parameter names, if necessary.
5. Write definition **body** by starting with a template.
6. Write additional **tests**, if needed, especially 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 pattern if you are having trouble with a problem.

Lecture 04 Summary

What happens next?

Over four lectures we will develop 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 the design pattern to write recursive functions following version 1 of the Rules of Recursion
- How to use **`zero?`** and **`sub1`** to write recursive functions
- How the Rules of Recursion (version 1) guarantee termination
- How to test recursive functions.
- How to use the stepper to measure efficiency.

L04: Allowed constructs

Newly allowed constructs:

```
sub1 zero?
```

Recursion (following the first version of the rules)

Previously allowed constructs:

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