Functions

CS135 Lecture 02

L02.0 Function definitions

Function definitions in mathematics

$$\begin{aligned} f(x) &= x^2 + 3x + 4 \\ g(x, y) &= x^2 + 6xy + y^2 + 9x - 3y - 100 \end{aligned}$$

An **application** of a function supplies **arguments** for the **parameters**, which are substituted into the algebraic expression:

$$g(2, 3) = 2^2 + 6 \cdot 2 \cdot 3 + 3^2 + 9 \cdot 2 - 3 \cdot 3 - 100 = -42$$

An argument is substituted each time the associated parameter is used. The arguments supplied may themselves be applications:

$$g(f(2), f(3)) = g(14, 22) = 2488$$

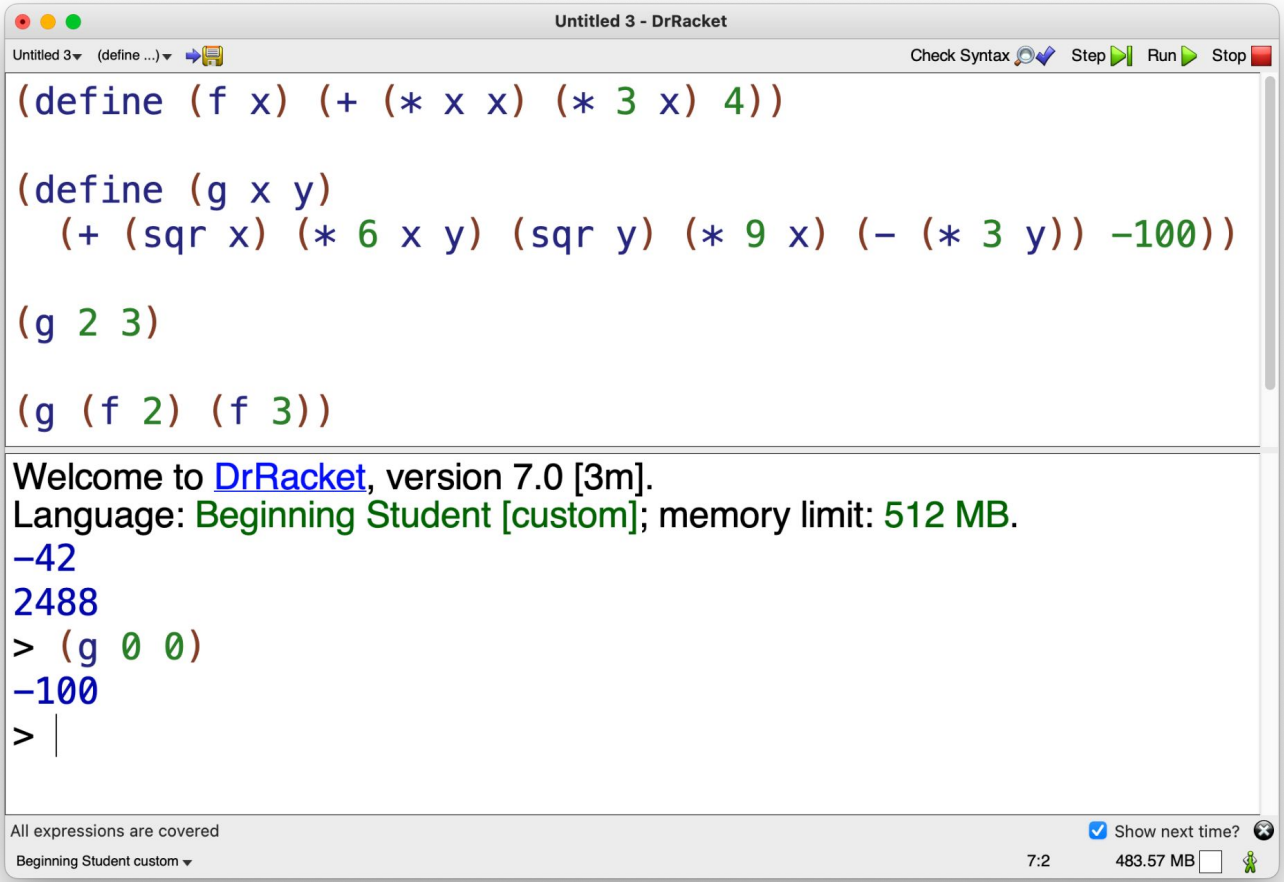
Translating to Racket with **`define`** and prefix notation

```
(define (f x) (+ (* x x) (* 3 x) 4))

(define (g x y)
   (+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100))

(g 2 3)

(g (f 2) (f 3))
```

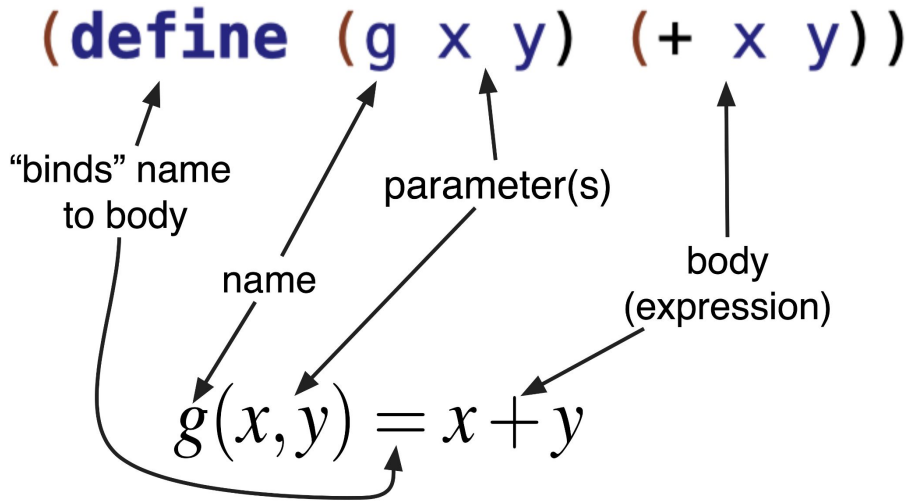

Untitled 3 - DrRacket
Untitled 3
(define ...)
Check Syntax
Step
Run
Stop
(define (f x) (+ (* x x) (* 3 x) 4))
(define (g x y)
(+ (sqr x) (* 6 x y) (sqr y) (* 9 x) (- (* 3 y)) -100))
(g 2 3)
(g (f 2) (f 3))
Welcome to DrRacket, version 7.0 [3m].
Language: Beginning Student [custom]; memory limit: 512 MB.
-42
2488
> (g 0 0)
-100
>
All expressions are covered
Show next time?
Beginning Student custom
7:2
483.57 MB

Defining functions

A function definition consists of:

- a **name** for the function,
- a list of **parameters**,
- a single **body** expression.

The body expression typically uses the parameters together with other built-in and user-defined functions.



Applying user-defined functions

1) An application of a user-defined function substitutes arguments for the corresponding parameters throughout the definition's expression.

```
(define (g x y) (+ x y))
```

The substitution for `(g 3 5)` would be `(+ 3 5)`.

2) All instances of a parameter in the body are replaced in a single step:

```
(define (h x y) (+ x x x y))
```

The substitution for `(h 10 9)` would be `(+ 10 10 10 9)`.

Substitution steps

```
(define (f x) (sqr x))
(define (g x y) (+ x y))

(g (g 1 3) (f 3))
⇒(g (+ 1 3) (f 3))
⇒(g 4 (f 3))
⇒(g 4 (sqr 3))
⇒(g 4 9)
⇒(+ 4 9)
⇒13
```

When faced with choices of substitutions:

1. Apply function definitions only when all arguments are simple values, like numbers and Boolean values.
2. When you have a choice, take the leftmost one.

Predicates

Functions that produce Bool values are called “predicates”.

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))
(define (hot? t) (> t 30))
(define (warm? t) (not (or (cold? t) (cool? t) (hot? t))))
```

By convention, the names of predicates end in a “**?**”.

Notice how **`cool?`** is defined in terms of **`cold?`**, and **`warm?`** is defined in terms of the other functions.

Identifiers

The identifiers that name constants, functions and parameters follow some rules:

- Identifiers can contain letters, numbers, `-`, `_`, `.`, `?`, `=`, and some other characters.
- Identifiers cannot contain space, brackets of any kind, or quotation marks like `` ` ' " ``.
- Identifiers must contain at least one non-number.

Identifiers should be meaningful, when possible.

`f`, **`x-ray`**, **`wHaTeVeR`**, **`hello!`**, and **`2d`** are all valid identifiers.

Observations

As with Mathematical functions…

Changing names of parameters does not change what the function does. These functions have the same behavior:

`(define (f x) (* x x))` and `(define (f z) (* z z))`

Different functions may use the same parameter name. There is no problem with:

`(define (f x) (* x x))` and `(define (g x y) (- x y))`

Parameter order matters. The following two functions are not the same:

`(define (g x y) (- x y))` vs. `(define (g y x) (- x y))`

L02.1 Substitution rules

Substitution rules

We now make our model of computation more formal by more precisely defining a set of **substitution rules**. The substitution process repeatedly simplifies the program. At each step, the result is a valid (but simpler) Racket program. It eventually simplifies to a value.

We are defining a mathematically oriented model of computation that is independent of any specific physical computer, but it still accuracy reflects abilities and limitations of physical computers.

We will add additional rules in the next lecture.

Rule 0: Application of built-in functions

A **substitution step** finds the **leftmost/inner subexpression,** with no parentheses inside, and rewrites it by replacing the subexpression by its value.

```
(+ (* 3 2) 5) ⇒ (+ 6 5) ⇒ 11
(expt 2 10) ⇒ 1024
```

Formally, the substitution rule is:

$(f\ v_1 \ldots v_n) \Rightarrow v$,

where f is a built-in function, *(except Boolean operators)*

$v_1 \ldots v_n$ are values,

and v is the value of $(f\ v_1 \ldots v_n)$.

Ellipses…

For built-in functions *f* with **one** parameter, the rule is:
$(f\ v_1) \Rightarrow v$, where *f* is a built-in function, v_1 is a value, and *v* is the value of $(f\ v_1)$

For built-in functions *f* with **two** parameters, the rule is:
$(f\ v_1\ v_2) \Rightarrow v$,
where *f* is a built-in function, v_1 and v_2 are values, and *v* is the value of $(f\ v_1\ v_2)$.

For built-in functions *f* with **three** parameters, the rule is:
$(f\ v_1\ v_2\ v_3) \Rightarrow v$,
where *f* is a built-in function; v_1, v_2, v_3 are values; and *v* is the value of $(f\ v_1\ v_2\ v_3)$.
…

We can't keep writing down rules forever, so we use ellipses in Rule 0 to show the pattern:
$(f\ v_1 \ldots v_n) \Rightarrow v$,
where *f* is a built-in function, $v_1 \ldots v_n$ are values, and *v* is the value of $(f\ v_1 \ldots v_n)$.

Rule 1: Application of Boolean operators

`(and true … false …)` ⇒ `false`	“short circuit” evaluation
`(and true … true)` ⇒ `true`	all arguments have value `true`
`(or false … true …)` ⇒ `true`	“short circuit” evaluation
`(or false … false)` ⇒ `false`	all arguments have value `false`

`(not true)` ⇒ `false`
`(not false)` ⇒ `true`

Here the ellipses are not showing patterns, but showing omissions.

Rule 2: Application of user-defined functions

(*f* v_1 ... v_n) ⇒ *e'*

where **(define (***f* x_1 ... x_n**)** ***e*****)** occurs to the left/above,
and *e'* is obtained by substituting into the expression *e*,
with all occurrences of the parameter x_i replaced by the value v_i ($1 \le i \le n$).

```
(define (f x y) (* x y (sqr y)))
(f (- 3 1) (+ 1 2))
⇒(f 2 (+ 1 2))
⇒(f 2 3)
⇒(* 2 3 (sqr 3))
⇒(* 2 3 9)
⇒ 54
```

Substitution steps

```
(define (cold? t) (< t 8))
(define (cool? t) (and (< t 16) (not (cold? t))))
(define (hot? t) (> t 30))
(define (warm? t) (not (or (cold? t) (cool? t) (hot? t))))
(warm? 32)
```

⇒ `(not (or (cold? 32) (cool? 32) (hot? 32)))`	*rule 2*
⇒ `(not (or (< 32 8) (cool? 32) (hot? 32)))`	*rule 2*
⇒ `(not (or false (cool? 32) (hot? 32)))`	*rule 0*
⇒ `(not (or false (and (< 32 16) (not (cold? 32)))(hot? 32)))`	*rule 2*
⇒ `(not (or false (and false (not (cold? 32)))(hot? 32)))`	*rule 0*
⇒ `(not (or false false (hot? 32)))`	*rule 1*
⇒ `(not (or false false (> 32 30)))`	*rule 2*
⇒ `(not (or false false true))`	*rule 0*
⇒ `(not true)`	*rule 1*
⇒ `false`	*rule 1*

Inexact numbers

Rational numbers are great if we just want to add, subtract, multiple, divide, and use integer exponents, but as soon as we write $2^{1/2} = \sqrt{2} = ?$ we discover we need irrational numbers as well, i.e. we need real numbers.

Unfortunately, irrational numbers can't be represented exactly in finite memory.

Instead, DrRacket uses an approximate representation of inexact numbers built into the physical hardware of the computer called "floating point" numbers.

```
> (sqrt 2)
#i1.4142135623730951
>
```

The `#i` tells us the result is inexact

Inexact numbers and the `Num` type

The range of an inexact number is roughly $\pm 2.23 \times 10^{-308}$ to $\pm 1.80 \times 10^{308}$

The precision of an inexact number is roughly 16 decimal digits

We say “roughly” because physical computers really work in binary (base 2) rather than decimal.

Inexact numbers can take on some unexpected values, including values that represent “Not a Number” (NaN) and “Infinity”. You shouldn’t see these.

When we say “number” in CS135, we mean it could be a natural number, an integer, a rational number, or an inexact number. When we want to indicate that a function could produce or consume any type of number, we write `Num`.

Built-in Math Functions

Racket has lots of built-in math functions that will be familiar to you.These all consume a **`Num`** and produce a **`Num`**.

`abs`	absolute value of a **`Num`**
`sqrt`	square root of a **`Num`**
`log`	base-**`e`** logarithm of a **`Num`**
`exp`	**`e`** raised to a **`Num`**
`cos`, **`sin`**, **`tan`**	trig functions consume an angle in radians
`acos`, **`asin`**, **`atan`**	the inverse trig functions produce an angle in radians

Racket also provide constant values **`pi`** and **`e`**.

The **`inexact?`** predicate determines if a **`Num`** is inexact.

Remember that inexact numbers are not exact

The mathematical properties you expect may not be true with inexact numbers.

```
> (sin pi)
#i1.2246467991473532e-16
> (= (sin pi) 0)
false
>
```

In particular, never test inexact numbers for equality.

$1.2246467991473532 \times 10^{-16} \neq 0$

Instead, check that numbers are “close enough”: `(< (abs (- x y)) 0.0001)`

L02.3 Comments

Writing comments in Racket

Comments let us write notes to ourselves or other programmers.

Comments start with a semicolon (`;`) and extend to the end of the line.

```
;; By convention, please use two semicolons, like
;; this, for comments which use a whole line.

;; Comments after code use one semicolon.
(* pi r r) ; computing the area of a circle
(define freezing 0)   ; freezing point of water
(define boiling 100)  ; boiling point of water
```

Formatting your assignment submissions

Each file you submit should start with a header to identify yourself, the term, the assignment and the problem. There is no specifically required format, but here's one acceptable way to format the header:

```
;;
;; ***********************************************
;; Chet G Peaty (12345678)
;; CS 135 Winter 2025
;; Assignment 03, Problem 4
;; ***********************************************
;;
```

Formatting your assignment submissions

If the assignment asks for multiple solutions in the same file, put them in the same order as the assignment, separated by a comment.

```
;;
;; Question 4. Part c.
;;
```

Each function should be preceded by a comment that describes its purpose.

```
;; (twice n) produces a value that is twice as large as n
(define (twice n) (* 2 n))
```

In addition, keep your line lengths no longer than 102 characters, the default maximum in DrRacket. Overly long lines are hard to read.

Block comments

Sometimes it's useful to "comment out" a section of a program. There are two options to do this quickly:

1) Select the text and use the *Racket → Comment Out with Semicolons* command

2) Use a multi-line comment:

```
#|
(define (function-to-temporarily-remove x y)
(+ x y)) |#
```

Never never use *Racket → Comment Out with a Box* or we won't be able to mark your assignment and that will make you unhappy.

Lecture 02 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 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.

L02: You should know

- How to define functions in Racket with **`define`**
- How to apply user-defined functions
- Predicates and the **`?`** convention
- Formal substitution rules for:
  - built-in functions, which include the operations we learned in Lecture 01;
  - Boolean expressions; and
  - user defined function
- Inexact numbers and types of numbers: **`Nat`** vs. **`Int`** vs. **`Rat`** vs. **`Num`**
- Math functions: **`abs acos asin atan cos exp log sin sqrt tan`**
- Math constants: **`e pi`**
- The **`inexact?`** predicate
- How to format your assignments for submission

L02: Allowed constructs

Newly allowed constructs:

`; abs acos asin atan cos define` *(functions)* `e exp inexact? log pi sin sqrt tan Num`

Previously allowed constructs:

`( ) + - * / = < > <= >=`

`and define` *(constants)* `expt false max min not or quotient remainder sqr true Bool Int Nat Rat`