Welcome to CS 115 (Winter 2020)

Web page (the main information source):

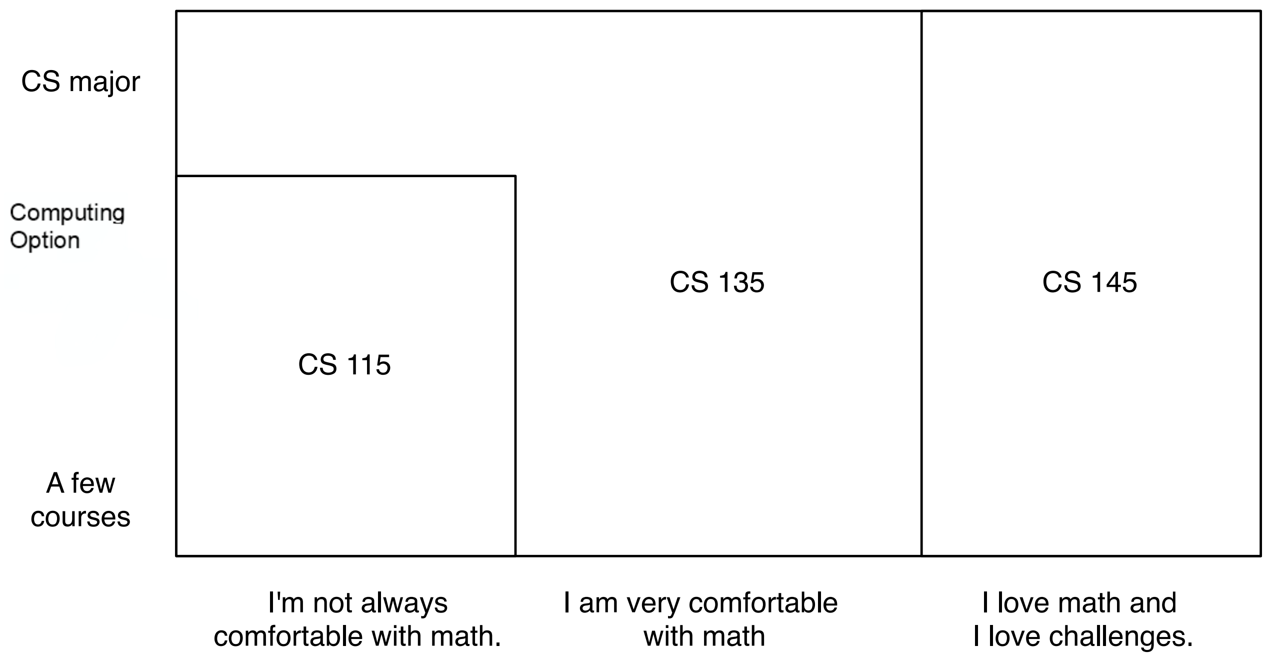
`https://www.student.cs.uwaterloo.ca/~cs115/`

Course Personnel: Contact information and office hours for all staff:

- instructors,
- ISAs (instructional support assistants),
- IAs (instructional apprentices), and
- the ISC (instructional support coordinator)

is available on the web page.

Am I in the right course?


CS major
Computing Option
A few courses
CS 115
CS 135
CS 145
I'm not always comfortable with math.
I am very comfortable with math
I love math and I love challenges.

Factors: interest in CS, comfort with math

Additional choice: CS 105 for non-math students

Course components

Lectures: Two 80 minutes lectures per week

Textbook: “How to Design Programs” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi, First Edition (`http://www.htdp.org/2003-09-26`) 2019

Presentation handouts: available on the web page and as a printed course pack from SCH Bookstore

Labs: 80 minutes using DrRacket v7.0 or higher (`https://download.racket-lang.org`)

Class Participation using Clickers

- There will be several multiple choice questions each lecture.
- You will earn 1 mark for any answer, plus 1 more if correct.
- Best 75% over term will be used to calculate grade.
- Questions **must** be answered in your own lecture.
- **Never** use another student's clicker.
- **You must register your clicker as part of A0.**
- *Purpose: To encourage active learning and provide real-time feedback (for the student and instructor).*

Assignments

- A0 - course logistics. **Must be completed for remaining assignments to be graded.**
- Assignments due weekly (roughly).
- You may use lab computers or your own computer.
- You must submit your assignments using MarkUs.
  - No email submissions will be accepted.
  - Submit your solutions early and often!
  - Basic tests are run on your submissions. Be sure to check the results.

Marking Scheme - check details on web page

- 20% Assignment
- 5% Participation
- 75% Exams - **You must pass the weighted exam average**
  - 30% Midterm
  - 45% Final

Under certain conditions, labs can earn an additional 3% bonus.

Grade Appeals: Review policy on course web page.

Academic Offenses: Review policy on course web page.

Work for your first week:

- Check your schedule for correct lecture and lab times.
- Attend lectures and lab.
- Read the Web pages for important course information (including staff, marking scheme, academic integrity policies, etc.).
- Read the Survival Guide (see the ”Resources” web page).
- Complete Assignment 0 (including registering your clicker).

Suggestions for success

- Keep up with the readings (keep ahead if possible).
- Attend lectures and take notes.
- Attend weekly labs and do the exercises.
- Start assignments early.
- Visit office hours to get help.
- Integrate exam study into your weekly routine.
- Keep on top of your workload.

More suggestions for success

- Follow our advice on approaches to writing programs (e.g. design recipe, templates).
- Go beyond the minimum required (e.g. do extra exercises).
- Maintain a “big picture” perspective: look beyond the immediate task or topic.
- Review your assignments and midterm: learn from your mistakes.
- Read email sent to your UW account.

Programming practice

Lectures prepare you for labs.

Labs prepare you for assignments.

Assignments prepare you for exams.

You must do your own work in this course; read the section on plagiarism in the CS 115 Survival Guide.

Role of programming languages

At the lowest level, a computer executes instructions in machine language. Machine language is designed for particular hardware, rather than being human-friendly and general.

People use high-level languages to express computation naturally and generally.

Programming language design

Imperative:

- frequent changes to data
- examples: machine language, Java, C++

Functional:

- computation of new values, not transformation of old ones
- examples: Excel formulas, LISP, ML, Haskell, Mathematica, XSLT, R (used in STAT 231)
- more closely connected to mathematics
- easier to design and reason about programs

Why start with Racket?

- used in education and research since 1975.
- minimal but powerful syntax
- tools easy to construct from a few simple primitives
- DrRacket, a helpful environment for learning to program
- nice transition to an imperative language, Python, in CS 116

Course goals

CS 115 **isn’t** a course in programming in Racket.

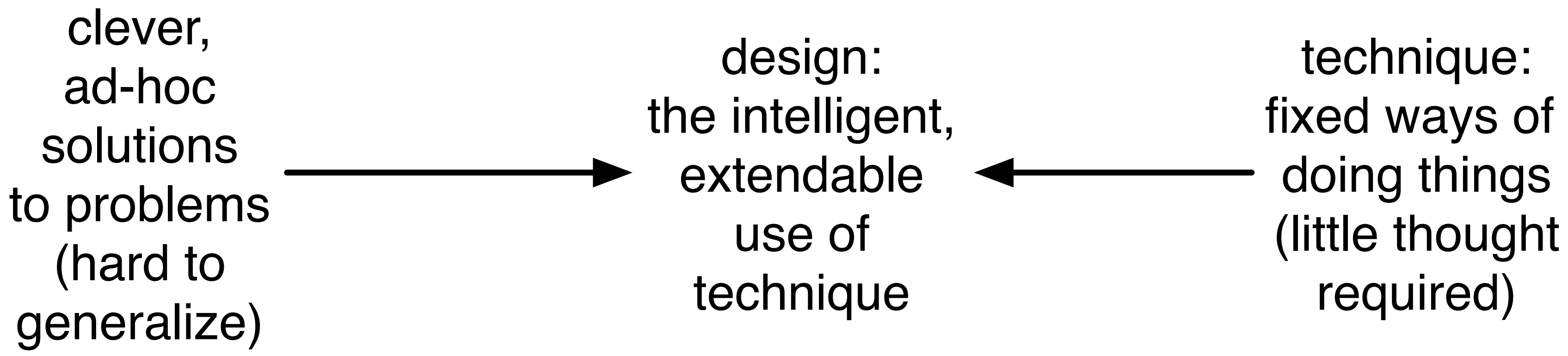
CS 115 **is** an introduction to computational thought (thinking precisely and abstractly).

CS 115 uses only a very small subset of Racket.

CS 116 will make the transition to an industry-oriented imperative language, Python.

Knowing two different programming paradigms will make it easier for you to extract the higher-level concepts being taught.

We will cover the whole process of designing programs.


clever, ad-hoc solutions to problems (hard to generalize)
design: the intelligent, extendable use of technique
technique: fixed ways of doing things (little thought required)

Careful use of design processes can save time and reduce frustration, even with the fairly small programs written in this course.

Themes of the course

- design (the art of creation)
- abstraction (finding commonality, not worrying about details)
- refinement (revisiting and improving initial ideas)
- syntax (how to say it), expressiveness (how easy it is to say and understand), and semantics (the meaning of what's being said)
- communication (in general)

Goal: Learning how to think about solving problems using a computer.

Background terminology

Values are numbers.

For example, $5, 4/9, 3.14159$.

Expressions combine values with operators and functions.

For example, $5 + 2, \log_2(2 \cdot 8)$, $\frac{\sqrt{2}}{100 \cdot 5}$. Expressions can be simplified to values.

Functions generalize similar expressions.

Functions in mathematics

A function generalizes similar expressions.

$3^2 + 4 \cdot 3 + 2$

$6^2 + 4 \cdot 6 + 2$

$7^2 + 4 \cdot 7 + 2$

These are generalized by the function
$q(x) = x^2 + 4 \cdot x + 2$.

A mathematical **function definition** consists of

- the name of the function,
- its **parameters**, and
- a mathematical expression using the parameters.

Examples:

$f(x) = x^2$

$g(x, y) = x - y$

A **function application (or use)** supplies **arguments** for the parameters. Examples:

$g(3, 1)$

$g(f(2), g(3, 1))$

The mathematical expression is **evaluated** by substitution. The arguments are substituted into the expression in place of the parameters.

Example:

$g(3, 1) = 3 - 1 = 2$

We say that g **consumes** 3 and 1 and **produces** 2.

Mathematical expressions

For some mathematical functions, function applications are written in a different way.

In arithmetic expressions, the symbols for addition, subtraction, and division are put between numbers, for example, $3 - 2 + 4/5$.

Precedence rules (division before addition, left to right) specify the order of operation.

To associate a function with arguments in a way different from that given by the rules, parentheses are required: $(6 - 4)/(5 + 7)$.

Racket values, expressions, and functions

The simplest Racket values are numbers.

- Integers in Racket are unbounded.
- Rational numbers are represented exactly.
- Irrational numbers can be approximated by inexact Racket values.

We will learn about other Racket values later.

Racket expressions are built from values, operators, and functions.

Function applications in Racket

In a Racket function application, parentheses are put around the function name and the arguments.

To translate from mathematics to Racket

- move '(' to before the name of the function, and
- replace each comma with a space.

$g(3, 1)$ becomes (g 3 1)

$g(f(2), g(3, 1))$ becomes (g (f 2) (g 3 1))

There is one set of parentheses for each function application.

Converting an expression into Racket

As before, we put parentheses around the function and arguments.

$3 - 2$ becomes `(- 3 2)`

$3 - 2 + 4/5$ becomes `(+ (- 3 2) (/ 4 5))`

$(6 - 4) \cdot (3 + 2)$ becomes `(* (- 6 4) (+ 3 2))`

In Racket, parentheses are used to associate arguments with functions, and are no longer needed for precedence.

Extra parentheses are harmless in mathematical expressions, but they are harmful in Racket. **Only use parentheses when necessary.**

The DrRacket environment

- designed for education, powerful enough for “real” use
- sequence of language levels keyed to textbook
- includes good development tools
- two windows: Interactions and Definitions
- Interactions window: a read-evaluate-print loop (REPL)

Built-in functions

Racket has many built-in functions, such as familiar functions from mathematics, other functions that consume numbers, and also functions that consume other types of data.

You can find the quotient of two integers using quotient and the remainder using remainder.

```
(abs −5)
(quotient 75 7)
(remainder 75 7)
```

Racket expressions causing errors

What is wrong with each of the following?

- `(* (5) 3)`
- `(+ (* 2 4)`
- `(5 * 14)`
- `(* + 3 5 2)`
- `(/ 25 0)`

Preventing errors

Syntax: the way we're allowed to express ideas

Syntax error when an expression cannot be interpreted using the syntax rules: `(+ - 3)`

Semantics: the meaning of what we say

Semantic error when an expression cannot be reduced to a value by application of substitution rules. It occurs when the expression is being simplified, after the syntax has been checked: `(/ 5 0)`

'Trombones fly hungrily' has correct syntax (spelling, grammar) but no meaning.

English syntax rules (e.g. a sentence has a subject, a verb, and an object) are not rigidly enforced.

Racket syntax rules are interpreted by a machine. They are rigidly enforced.

If you break a syntax rule, you will see an error message.

Syntax rule: a **function application** consists of the function name followed by the expressions to which the function applies, all in parentheses, or (functionname exp1 . . . expk).

We need rules for function names and expressions to complete this.

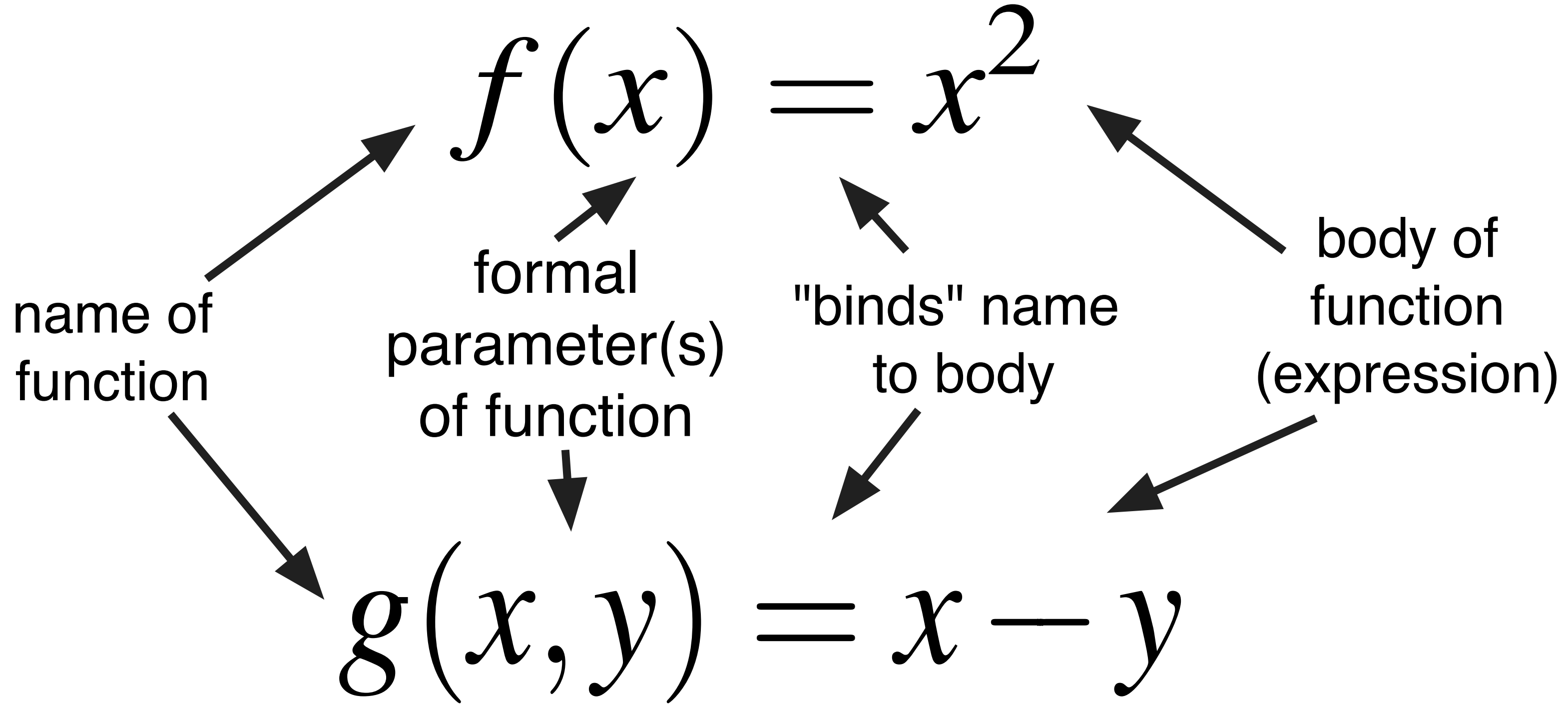
So far Racket is no better than a calculator or a spreadsheet.

A spreadsheet formula applies functions to data.

A spreadsheet application provides many built-in functions, but typically it is hard to construct your own.

Racket provides a modest number of built-in functions, but makes it easy to construct your own.

Defining functions in mathematics


name of function
formal parameter(s) of function
"binds" name to body
body of function (expression)

$$f(x) = x^2$$

$$g(x, y) = x - y$$

Important observations:

- Changing names of parameters does not change what the function does: $f(x) = x^2$ and $f(z) = z^2$ have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: $g(3, 1) = 3 - 1$ but $g(1, 3) = 1 - 3$.
- Applying a function creates a new value.

Defining functions in Racket

Our definitions $f(x) = x^2$ and $g(x, y) = x - y$ become

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

define is a **special form**; it looks like a Racket function, but not all of its arguments are evaluated.

It **binds** a name to an expression (which uses the parameters that follow the name).

A function definition consists of

- a name for the function,
- a list of parameters, and
- a single “body” expression.

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

To give names to the function and parameters, we use identifiers.

Beginner Syntax rule: an **identifier** starts with a letter, and can include letters, numbers, hyphens, underscores, and a few other punctuation marks.

It cannot contain spaces or any of ( ) , { } [ ] ‘ ’ “ ”.

Syntax rule: **function definition** is of the form
(define (id1 . . . idk) exp), where exp is an expression and each id is an identifier.

We can adjust the syntax rule for a function application to say that a function name is a built-in name or an identifier.

Important observations:

- Changing names of parameters does not change what the function does: (define (f x) (∗ x x)) and (define (f z) (∗ z z)) have the same behaviour.
- The same parameter name can be used in different functions, as in f and g.
- The order of arguments must match the order of the parameters in the definition of the function: (g 3 1) produces 2 but (g 1 3) produces -2.

DrRacket’s Definitions window

- can accumulate definitions and expressions
- Run button loads contents into Interactions window
- can save and restore Definitions window
- provides a Stepper to evaluate expressions step-by-step
- features include: error highlighting, subexpression highlighting, syntax checking, bracket matching

Constants in Racket

The definitions $k = 3$ and $p = k^2$ become

```
(define k 3)
(define p (* k k))
```

The first definition binds the identifier k to the value 3.

In the second definition, the expression (* k k) is first evaluated to give 9, and then p is bound to that value.

Syntax rule: a **constant definition** is of the form (define id exp).

Defining constants allows us to give meaningful names to values (e.g. interest-rate, passing-grade, and starting-salary).

Usefulness of constants:

- Make programs easier to understand.
- Usually make future changes easier.
- May reduce typing (but may not).

The text uses the term variable to refer to constants, but their values cannot be changed.

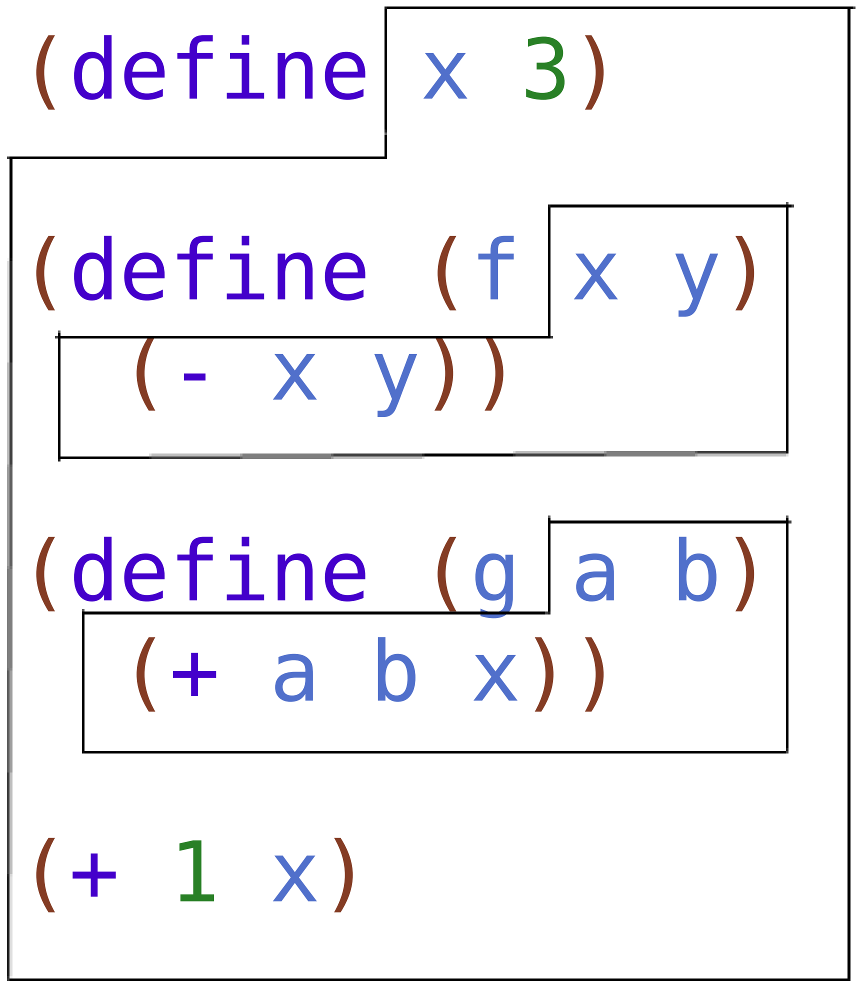
Programs in Racket

A **Racket program** is a sequence of definitions and expressions.

- The definitions are of functions and constants.
- The expressions typically involve both user-defined and built-in functions and constants.

Programs may also make use of **special forms** (which look like functions, but don’t necessarily evaluate all their arguments).

Scope: where an identifier can be used



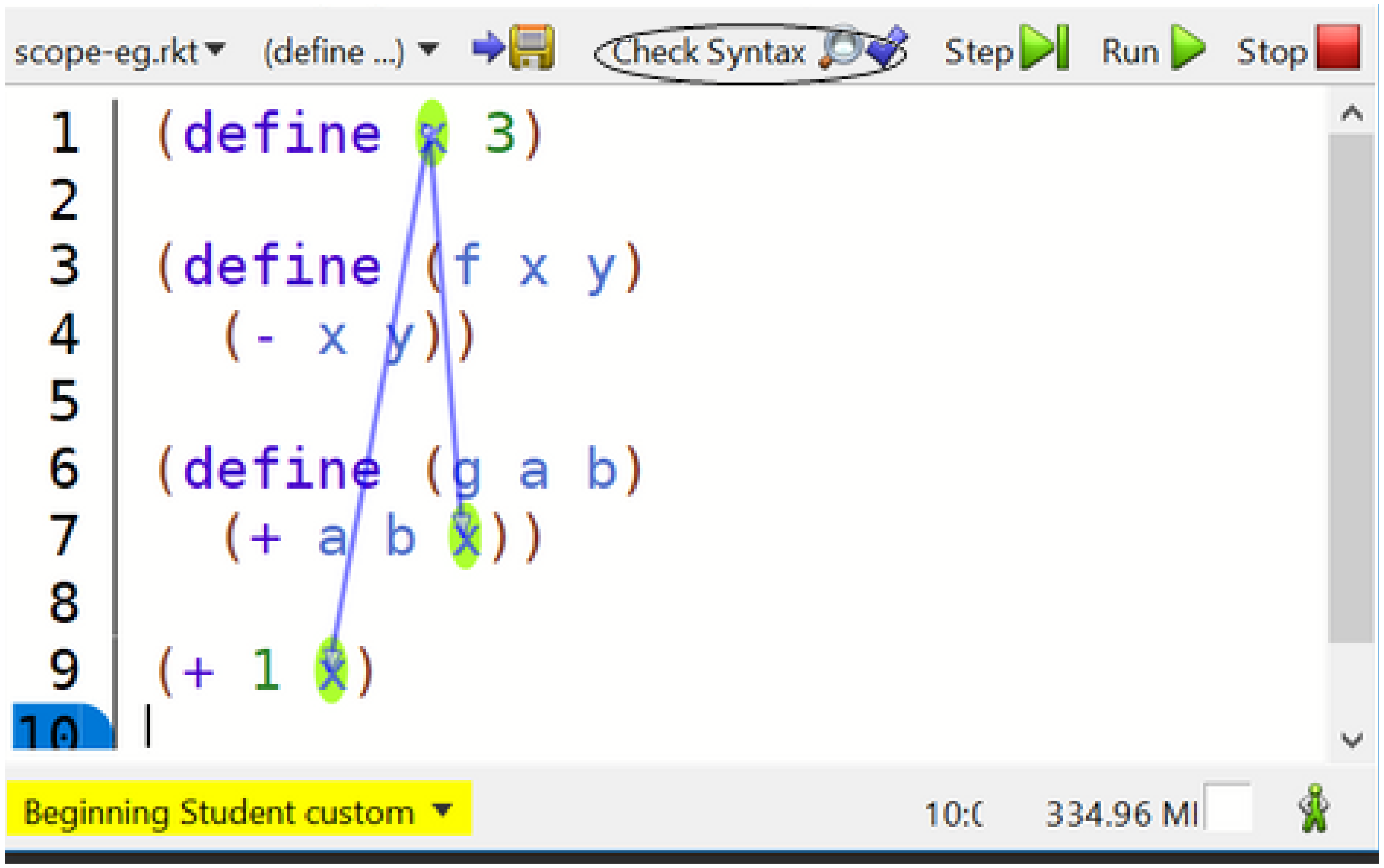
```
(define x 3)

(define (f x y)
   (- x y))

(define (g a b)
   (+ a b x))

(+ 1 x)
```

Scope in DrRacket



```
(define x 3)

(define (f x y)
  (- x y))

(define (g a b)
  (+ a b x))

(+ 1 x)
```

Identifiers and binding

Which of the following uses of x are allowed?

Can two functions use the same parameter name?

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

Can a constant and a function parameter use the same name?

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

Can two constants have the same name?

```
(define x 4)
(define x 5)
```

Can a constant and a function have the same name?

```
(define x 4)
(define (x y) (* 5 y))
```

Can the name of one function be the same as the parameter of another function?

```
(define (x y) (* 5 y))
(define (z x) (* 3 x))
```

Can a function name also be the name of one of its parameters?

```
(define (x x) (+ 1 x))                    ; Poor style
```

The importance of semantics

The English sentence “Cans fry cloud” is syntactically correct but has no semantic interpretation.

The English sentences “Students hate annoying professors”, “I saw her duck”, “You will be very fortunate to get this person to work for you”, and “Kids make nutritious snacks” are all ambiguous; each has more than one semantic interpretation.

Racket programs are unambiguous because of the rules of the language.

A semantic model

A semantic model of a programming language provides a method of determining the result of running any program.

A Racket program is a sequence of definitions and expressions.

Our model involves the simplification of the program using a series of steps (**substitution rules**).

For now, each step yields a valid Racket program.

A fully-simplified program is a sequence of definitions and values.

Values and Expressions

A Racket **value** is something which cannot be further simplified.

So far, the only things which are values are numbers. As we move through CS115, we will see many other types of values.

For example, 3 is a value, but (+ 3 5) and (f 3 2) are not.

A Racket **expression** is a value, a constant, or a function application.

Using substitution

We used substitution to evaluate a mathematical expression:

$$g(f(2), g(3,1)) = g(4, g(3,1)) = g(4,2) = 2$$

There are many possible ways to substitute, such as

$g(f(2), g(3,1)) = f(2) - g(3,1)$ and

$g(f(2), g(3,1)) = g(f(2), 2)$.

We will also use substitution for the Racket program:

```
(define (f x) (* x x))
(define (g x y) (− x y))
(g (f 2) (g 3 1))
```

Evaluating Racket values and constants

Racket values are already evaluated. No further steps are needed.

Racket constants evaluate to their simplified values.

Note that the value associated with a constant is determined when its definition is evaluated.

```
(define tax-rate 0.13)
(define price-factor (+ 1 tax-rate))
```

The expression (+ 1 tax-rate) is only evaluated once - when price-factor is defined.

Evaluating a Racket function application

Goal: a single sequence of substitution steps for any expression.

Constraints:

1. We first evaluate the arguments, and then apply the function to the resulting values.

2. When we have the choice among two or more substitutions, we take the **leftmost** (first) one.

These decisions do not change the final result.

We use the ‘yields’ symbol $\Rightarrow$ to show the result of a single step.

Substitution rules

Built-in function application: use mathematics rules.

```
(+ 3 5) ⇒ 8
(quotient 75 7) ⇒ 10
```

Constant: replace the identifier by the value to which it is bound.

```
(define x 3)
(* x (+ x 5))
⇒ (* 3 (+ x 5))
⇒ (* 3 (+ 3 5))
⇒ (* 3 8) ⇒ 24
```

Value: no substitution needed.

User-defined function application: for a function defined by (define (f x1 x2 . . . xn) exp), simplify a function application (f v1 v2 . . . vn) by replacing all occurrences of the parameter xi by the value vi in the expression exp.

Tricky points to remember:

- Each vi must be a **value**.
- Replace **all** occurrences in one step.

General rule: Anything that has been fully evaluated (e.g. a function definition) does not need to be repeated at the next step.

```
(define (f x) (* x x))
(define (g x y) (- x y))
(g (f 2) (g 3 1))
⇒ (g (* 2 2) (g 3 1))
⇒ (g 4 (g 3 1))
⇒ (g 4 (- 3 1))
⇒ (g 4 2)
⇒ (- 4 2)
⇒ 2
```

Tracing a program

Tracing: a step-by-step simplification by applying substitution rules.

Any expression that has been fully simplified does not need to appear in the next step of the trace.

Use tracing to check your work for semantic correctness.

If no rules can be applied but the program hasn't been simplified, there is an error in the program, such as (sqr 2 3).

Be prepared to demonstrate tracing on exams.

The Stepper may not use the same rules. Write your own trace to be sure you understand your code.

```
(define (term x y) (* x (sqr y)))
(term (- 5 3) (+ 1 2))
⇒ (term 2 (+ 1 2))
⇒ (term 2 3)
⇒ (* 2 (sqr 3))
⇒ (* 2 9)
⇒ 18
```

Goals of this module

You should be comfortable with these terms: function, parameter, application, argument, variable, expression, consume, produce, syntax, semantics, special form, bind, identifier, Racket program.

You should be able to define and use simple arithmetic functions, and to define constants.

You should know how to form expressions properly, and what DrRacket might do when given an expression causing an error.

You should understand the purposes and uses of the Definitions and Interactions windows in DrRacket.

You should be able to argue that your Racket code is syntactically correct, referring to Racket syntax rules.

You should understand the basic syntax of Racket, including rules for function application, function definition, and constant definition.

You should be able to trace the substitutions of a Racket program, using substitution rules for function applications and uses of constants.