Welcome to CS 135 (Fall 2019)

Instructors: Byron Weber Becker, Charles Clarke, Mark Giesbrecht, Dan Holtby, Kevin Lanctot, Adrian Reetz

Other course personnel: see website for details

- ISAs (Instructional Support Assistants)
- IAs (Instructional Apprentices)
- ISC (Instructional Support Coordinator)

Web page (main information source):
https://www.student.cs.uwaterloo.ca/~cs135/
Who Am I

• Kevin Lanctot, kevin.lanctot@uwaterloo.ca

• Office: DC 2131 (near the sky walk to MC and M3)

• Things to know about me:
  – I’m a talker not a typer.
  – I only check my email a few times a day.
  – Last name is pronounced long-k toe, i.e. “long toe” with the “k” sound after long.

• Note: lecture material, assignments, and exams are the same across all sections of CS135.
Additional Notes

• Besides the “official” course notes I will create some additional slides.

• They will be merged into the official slides but have “Additional Notes” and a slide number like 1.2 at the bottom.

• They will be available at the course web site https://www.student.cs.uwaterloo.ca/~cs135/cc/instructor_materials/

• Until the web site gets finalized, they’ll also be available at https://learn.uwaterloo.ca/ under the Content tab of CS135 by 9 pm the evening before each lecture.
Themes of the course

• Design (the art of creation)

• Abstraction (finding commonality, neglecting 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)

The approach is: learn how to think about solving problems using a computer.
Themes of the course

• One of our goals is to develop good programming habits.

• May seem unimportant for small or simple programs but the complexity will eventually become overwhelming.

• e.g.
  https://informationisbeautiful.net/visualizations/million-lines-of-code/
Lectures

Tuesdays and Thursdays 8:30 am – 9:50 am.

**Textbook:** “How to Design Programs (First Edition)” (HtDP) by Felleisen, Flatt, Findler, Krishnamurthi (find link on web site)

**Presentation handouts:** available on Web page and as printed coursepack from media.doc (MC 2018)

**Participation marks:** to encourage active learning
Participation marks: to encourage active learning

• Based on “clickers” (purchase at the Bookstore; register in A0) and several multiple-choice questions in each lecture

• One mark for any answer; second mark for correct answer

• We use the best 75% across the entire term to calculate 5% of the final mark.

• No sharing clickers: each clicker must be used by only one student in CS 135.

• No bringing your friend’s clicker to class, of course.

• You must attend the section you’re officially registered in in order to get credit for your participation.
Tutorials

• 16 sections on Fridays (starting tomorrow but attendance for this one is optional)

• Reinforces lectures with additional examples and problem-solving sessions

• Often directly applicable to the upcoming assignment

• Take your laptop and clicker

You should definitely be attending if your assignment marks are below 80%.
Assignments

**Timing:** About 10 assignments, typically due Tuesday at 9:00PM

**Software:** DrRacket v7.3 ([http://racket-lang.org](http://racket-lang.org))

**Computer labs:** MC 3003, 3004, 3005, 3027, 2062, 2063. Available for your use, but no scheduled labs. Most students use their own computers.

**A0:** Due *Tuesday, Sept 10.* Must complete before you are allowed to submit any subsequent assignment

**Submission:** Using MarkUs. More in A0. Submit early and often. No late submissions. No email submissions.
Exams

• Two midterms
  – Sept 30, 7:00–8:20 pm (80 minutes long)
  – Nov 4, 7:00–8:50 pm (110 minutes long)

• One final (date to be determined by the Registrar)

Do not make holiday travel plans before you know the date of all your final exams AND take into account the snow dates.
Marking scheme

• 20% Assignments (roughly weekly)
• 10% Midterm 1
• 15% Midterm 2
• 50% Final exam
• 5% Participation (on best 75% of the clicker questions)

To pass the course:

⇒ Your weighted assignment average must be 50% or greater.
⇒ Your weighted exam average must be 50% or greater.
Getting help

• Tutors have regular office hours. Schedule on web site.

• Instructors also have office hours.

• **Piazza**: An on-line forum where you can ask questions, other students and instructors answer them.
  
  – Regularly check the official assignment pinned posts
  
  – Use meaningful subject headings (not just “A3 problem”; what’s your specific problem?)
  
  – Search previous posts before posting; **Don’t duplicate!**
  
  – Possible to post privately if necessary
Suggestions for success

Read the CS135 Survival Guide as soon as possible. Find it on the course web site under “Help”.

- Keep up with your assignments. Start them early. **This is key!**
- Go over your assignments and exams; learn from your mistakes.
- Attend lectures; take notes
- Visit office hours as needed; earlier is better.
- Follow our advice on approaches to writing programs (e.g. design recipe, templates).
- Read your mail sent to your UW email account.
Suggestions for success (cont.)

- Keep up with the readings (keep ahead if possible).
- Integrate exam study into your weekly routine.
- Go beyond the minimum required (e.g. do extra exercises).
- Maintain a “big picture” perspective: look beyond the immediate task or topic.
Academic integrity

- You must do your own work.
- Policy 71 - Student Discipline: plagiarism, sharing assignments, etc.
- Running out of time? It is better to hand in a partial assignment or nothing than to hand in someone else’s work.
- Be careful about posting code to Piazza. If it looks like it could have come from your assignment, don’t post it (publicly).
Academic Integrity

• Do your own work. ⇒ Do not copy another student’s work.
  – You may talk to other people about how to do an assignment, but do not write or record anything.
  – You cannot view or share another person’s code.

• Standard penalty for first offence is 0 on the assignment and -5% on the final grade.

• Suspension/expulsion for 2nd offence in any course.

• We have software that detects similar code.
Intellectual property

The teaching material used is CS 135 are the property of its authors. This includes:

• Lecture slides and instructor written notes
• Assignment specifications and solutions
• Tutorial slides and notes
• Examinations and solutions

Sharing this material without the IP owner’s permission is a violation of IP rights.
Goals of this module

You should understand how the course is organized [1–7].

You should be familiar with the course resources available to you [9].

You should know what you need to do to earn the mark you desire [8,10,11].

You should know how to avoid plagiarism [12].
Module 02: Functions

Readings:

• HTDP, sections 1-3
• Survival and Style guides

Topics:

• Programming language design [1-4]
• The DrRacket environment [5-6]
• Values, expressions, & functions [7-21]
• Defining functions [22-30]
• Scope [31]
• Programming in DrRacket [32-34]
Programming language design

**Imperative**: based on frequent changes to data
- Examples: machine language, Java, C++, Turing, VB

**Functional**: based on the computation of new values rather than the transformation of old ones.
- Examples: Excel formulas, LISP, ML, Haskell, Erlang, F#, Mathematica, XSLT, Clojure.
- More closely connected to mathematics
- Easier to design and reason about programs
Racket

- a functional programming language
- minimal but powerful syntax
- small toolbox with ability to construct additional required tools
- interactive evaluator
- used in education and research since 1975
- a dialect of Scheme
- graduated set of teaching languages are a subset of Racket
Functional vs. imperative

Functional and imperative programming share many concepts. However, they require you to think differently about your programs. If you have had experience with imperative programming, you may find it difficult to adjust initially.

By the end of CS 136, you will be able to express computations in both these styles, and understand their advantages and disadvantages.
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 (now) and Definitions (later)

• Interactions window: a read-evaluate-print loop (REPL)
Setting the language in DrRacket

CS 135 will progress through the Teaching Languages starting with *Beginning Student*.

1. Under the *Language* tab, select *Choose Language* ...

2. Select * Beginning Student * under * Teaching Languages*

3. Click the *Show Details* button in the bottom left

4. Under *Constant Style*, select *true false empty*

Remember to follow steps 3 and 4 each time you change the language.
Values, expressions, & functions

Values are *numbers or other mathematical objects.*
Examples: $5, \frac{4}{9}, \pi$.

Expressions *combine values with operators and functions.*
Examples: $5 + 2, \sin(2\pi), \frac{\sqrt{2}}{100\pi}$.

Functions *generalize similar expressions.*
Example...
Values, expressions, & functions (cont)

Values are numbers or other mathematical objects.

Expressions combine values with operators and functions.

Functions generalize similar expressions.

Example:

\[ 3^2 + 4(3) + 2 \]
\[ 6^2 + 4(6) + 2 \]
\[ 7^2 + 4(7) + 2 \]

are generalized by the function

\[ f(x) = x^2 + 4x + 2. \]
Functions in mathematics

Definitions: \( f(x) = x^2, g(x, y) = x + y \)

Function definitions in mathematics consist of three components:

1. the name of the function (e.g. \( g \))

2. its parameters (e.g. \( x, y \))

3. an algebraic expression using the parameters as placeholders for values to be supplied in the future
Function application

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

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

An example: \( g(1, 3) = 1 + 3 = 4 \)

The arguments supplied may themselves be applications.

Example: \( g(g(1, 3), f(3)) \)
Function application (cont)

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

We evaluate each of the arguments to yield values.

Evaluation by substitution:

\[
g(g(1, 3), f(3)) = \]
\[
g(1 + 3, f(3)) = \]
\[
g(4, f(3)) = \]
\[
g(4, 3^2) = \]
\[
g(4, 9) = 4 + 9 = 13
\]
Many possible substitutions

Definitions: \( f(x) = x^2 \), \( g(x, y) = x + y \)

There are many mathematically valid substitutions:

\[
g(g(1, 3), f(3)) = g(1 + 3, f(3)) \ldots
\]

\[
g(g(1, 3), f(3)) = g(g(1, 3), 3^2) \ldots
\]

\[
g(g(1, 3), f(3)) = g(1, 3) + f(3) \ldots
\]

We’d like a **canonical form** (**a standard form**) for two reasons:

- Easier for us to think about
- When we extend this idea to programming, we’ll find cases where different orderings result in different values
Canonical substitutions

There are two rules for canonical substitutions

1. Functions are applied to values

2. When there is a choice of possible substitutions, always take the leftmost choice.

Now, for any expression:

• there is at most one choice of substitution;

• the computed final result is the same as for other choices.
The use of parentheses: ordering

In arithmetic expressions, we often place operators between their operands.

Example: $3 - 2 + 4/5$.

We have some rules (division before addition, left to right) to specify order of operation.

Sometimes these do not suffice, and parentheses are required.

Example: $(6 - 4)/(5 + 7)$.
The use of parentheses: functions

If we treat infix operators (+, −, etc.) like functions, we don’t need parentheses to specify order of operations:

Example: $3 − 2$ becomes $−(3, 2)$

Example: $(6 − 4)/(5 + 7)$ becomes $/(-(6, 4), +(5, 7))$

The substitution rules we developed for functions now work uniformly for functions and operators.

Parentheses now have only one use: function application.
The use of parentheses: functions

Racket writes its functions slightly differently: the function name moves inside the parentheses, and the commas are changed to spaces.

Example: \( g(1, 3) \) becomes \((g \ 1 \ 3)\)

Example: \( (6 - 4)/(5 + 7) \) becomes \((/ \ (- \ 6 \ 4) \ (+ \ 5 \ 7))\)

These are valid Racket expressions (once \( g \) is defined).

Functions and mathematical operations are treated exactly the same way in Racket.
Expressions in Racket

$3 - 2 + \frac{4}{5}$ becomes $(+ (− 3 2) (/ 4 5))$

$(6 - 4)(3 + 2)$ becomes $(∗ (− 6 4) (+ 3 2))$

Extra parentheses are harmless in arithmetic expressions. They are harmful in Racket.

*Only use parentheses when necessary* (to signal a function application or some other Racket syntax).
Infix vs. Prefix Notation

• **Infix notation** *operators go between values*: 1 + 2.
  – need precedence rules to determine order of evaluation
  – BEDMAS: brackets, exponents, division/multiplication, addition/subtraction
  – if operators have the same precedence (e.g. addition/subtraction) then evaluation left to right

• **Prefix notation**: *operators go before values*: (+ 1 2)
Evaluating a Racket expression

*We use a process of substitution*, just as with our mathematical expressions.

Each step is indicated using the *yields symbol* $\Rightarrow$.

$$(\ast (\neg 6 4) (\add 3 2))$$

$\Rightarrow (\ast 2 (\add 3 2))$

$\Rightarrow (\ast 2 5)$

$\Rightarrow 10$
Numbers in Racket

• Integers in Racket are *unbounded*, i.e. they can be any size the computer can handle.

• Rational numbers are *represented exactly* as a fraction which is why \( \frac{1}{2} \) is a value not an expression.

• Expressions whose *values are not rational numbers* are flagged as being *inexact*.

Example: \((\text{sqrt } 2)\) evaluates to \#i1.414213562370951.

We will not use inexact numbers much (if at all).
Expressions in Racket

Racket has many built-in functions which can be used in expressions:

- **Arithmetic operators**: +, −, ∗, /
- **Constants**: e, pi
- **Functions**: (abs x), (max x y . . .), (ceiling x) (expt x y), (exp x), (cos x), ...

Look in DrRacket’s “Help Desk”. The web page that opens has many sections. The most helpful is under **Teaching**, then “How to Design Programs Languages”, section 1.5.
Racket expressions causing errors

What is wrong with each of the following?

• (5 * 14)

• (* (5) 3)

• (+ (* 2 4)

• (* + 3 5 2)

• (/ 25 0)
Defining functions (in mathematics)

\[ f(x) = x^2 \]
\[ g(x, y) = x + y \]
Defining functions (in Racket)

\[
\text{(define (g x y) (+ x y))}
\]

(name of function)  \(g\)  \(x\), \(y\)  (formal parameter(s) of function)

"binds" name to body

body of function

\[
g(x, y) = x + y
\]
Defining functions (in Racket)

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

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

define is a **special form** in Racket (i.e. it looks like a Racket
function, but not all of its arguments are evaluated).

It **binds** (i.e. *associates or pairs up*) a name to an expression (which
uses the parameters that follow the name).
Defining functions (in Racket)

A function definition in Racket consists of:

1. a name for the function,
2. a list of parameters,
3. a single body expression which is an expression.

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

As pointed out in the previous few slides, a function definition in Racket has the same components (in a slightly different format) as function definition in mathematics.
Applying user-defined functions in Racket

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).
Applying user-defined functions in Racket

When faced with choices of substitutions, we use the same rules defined earlier: (1) apply functions only when all arguments are values; when you have a choice, take the leftmost one.

\[
g(g(1, 3), f(3)) = g(4, 3^2) = g(4, 9) = 4 + 9 = 13
\]
Applying user-defined functions in Racket

Each parameter name has meaning only within the body of its function. I.e. the two uses of x are independent.

(define (f x y) (+ x y))
(define (g x z) (* x z))

Additionally, the following two function definitions define the same function:

(define (f x y) (+ x y))
(define (f a b) (+ a b))
Defining constants

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

```
(define k 3)
(define p (sqr k))
```

The *effect* of `(define k 3)` is *to bind the name k to the value* 3.

In `(define p (sqr k))`, the expression `(sqr k)` is first evaluated to give 9, and then `p` is bound to that value.
Advantages of constants

• Can give *meaningful names* to useful values (e.g. *interest-rate*, *passing-grade*, and *starting-salary*).

• *Reduces errors* when such values need to be changed

• Makes programs *easier to understand*

Constants can be used in any expression, including the body of function definitions

Sometimes (incorrectly) called variables, but their values cannot be changed (until CS 136)
Scope

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

- The smallest enclosing scope has priority
- Can’t duplicate identifiers within the same scope

```scheme
(define x 3)
(define (f x y)
    (— x y ))
(define (g a b)
    (+ a b x))
(+ 1 x)
```

Racket Error: f: this name was defined...
Programming in DrRacket

The definitions window:

• Can save and restore your work to/from a file
• Can accumulate definitions and expressions
• Run button loads contents into Interactions window
• Provides a Stepper to let one evaluate expressions step-by-step
• Features: error highlighting, subexpression highlighting, syntax checking
• Can check the scope of a constant...
DrRacket can show what expression each identifier is bound to.
Programs in Racket

A Racket program is a *sequence of definitions and expressions*. To run a program, *the expressions are evaluated, using substitution, to produce values*. Expressions may also make use of *special forms* (e.g. `define`), which look like functions, but don’t necessarily evaluate all their arguments.
Goals of this module

You should understand the basic syntax of Racket, how to form expressions properly, and what DrRacket might do when given an expression causing an error.

You should be comfortable with these terms: function, parameter, application, argument, constant, expression.

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

You should understand the purposes and uses of the Definitions and Interactions windows in DrRacket.
Module 02 Summary

Some of the Components of Racket

1. **Values** are numbers or other mathematical objects. [7]
   Examples: $5$, $\frac{4}{9}$ (a rational number), $\pi$.

2. **Expressions** combine values with operators and functions. [7]
   Examples: $5 + 2$, $\sin(2\pi)$, $\frac{\sqrt{2}}{100\pi}$.

3. **Functions** generalize similar expressions. [8]

4. **Function definitions** consist of a name, parameters and an algebraic expression, e.g. `(define (f x y) (+ x y))`. [9]
Module 02 Summary

How to Use a Function

5. An **application** of a function supplies **arguments** for the **parameters**, which are substituted into the algebraic expression. [10]

6. Functions are evaluated by **substitution**. [11]
   (a) Functions are applied to **values**. [13]
   (b) When there is a **choice** of possible substitutions, always take the **leftmost choice**. [13]
   (c) **Result**: This substitution process ensures there is no ambiguity in the meaning of a program.
Module 02 Summary

Parenthesis, Yields and Special Forms

7. Do not include extra parentheses in an expression. [17]

8. The *yields symbol*, \( \Rightarrow \), indicates a step in the substitution process. [18]

9. A *special form* looks like a Racket function but not all of its arguments are evaluated. [24]

10. The special form *define binds* (i.e. associates) a function name with a function body. [24]
Module 02 Summary

Parameters, Scope and Constants

11. Each parameter name has meaning only within the body of its function. [28]

12. The special form `define` can also be used to define constants. [29]

13. Constant names can be arbitrary, so pick meaningful ones. [30]

14. An identifier’s `scope` is where it has effect. [31]
   (a) The smallest enclosing scope has priority. [31]
   (b) Can’t duplicate identifiers within the same scope. [31]
Module 02 Summary

A Racket Program

15. DrRacket has a **Definitions Window** to store definitions and an **Interactions Window**. [32]

16. A Racket **program** consists of a sequence of definitions and expressions. [34]

17. To run a program, the expressions are evaluated, using substitution, to produce values. [34]
Module 03: The Design Recipe

Readings:

• HtDP, section 2.5
• Survival and Style Guides

Topics:

• Programs as communication
• The design recipe
• Using the design recipe
• Tests
• Contracts
Programs as communication

*Every program is an act of communication:*

- Between you and the computer
- Between you and yourself in the future
- Between you and others

Human-only comments in Racket programs: from a semicolon (;) to the end of the line.
Some goals for software design

Programs should be:

compatible, composable, correct, durable, efficient, extensible, flexible, maintainable, portable, readable, reliable, reusable, scalable, usable, and useful.
Some goals for software design

We emphasize the following...

- **Correctness**: Does it give the right output? Does it meet the specification? ⇒ starting now.
- **Efficiency**: Does it minimizing the use of computer resources such as processor time or memory usage? ⇒ starting 2nd year
- **Readability**: Can another programmer easily understand it?
- **Reliability**: Does it crash? Does it always work?
- **Flexibility**: Is it easy to change?
- **Extensibility**: Is it easy to add new features?
The design recipe

• Use it for every function you write in CS 135.

• A development process that leaves behind written explanation of the development

• Results in a trusted (tested) function which future readers (you or others) can understand
The design recipe

• For simple programs it almost seems like you do not need to go through all this effort.

• At some point: maybe in this course, maybe in 2nd year, you’ll reach a point where you start making mistakes because the complexity of the program is too large.

• Develop good habits now.
The five design recipe components

**Purpose:** Describes *what* the function is to compute.

**Contract:** Describes what *type of arguments* the function consumes and what *type of value it produces*.

**Examples:** Illustrating the *typical use* of the function.

**Definition:** The Racket definition of the function (*header & body*).

**Tests:** A *representative set of function applications* and their expected values. Examples also serve as Tests.
The design recipe

• The **Purpose** is more of a *description in English* whereas the **Contract** which is more of a *mathematical description*.

• **Consume** describes the parameters.

• **Produce** is what it returns, i.e. the result of applying the function.

• The **header** is the function name, parameters, and the *define* keyword.

• The **body** is the expression that computes the result.
The design recipe

• If you want help from course staff for handling problems with your code, you need to provide them with the functions contract, purpose, and examples.

• We will be using a new function: check-expect checks to see if a function has returned the expected results.

• Generally expressions can only use functions defined before the expression. This is not the case for check-expect.
Order of execution

The order in which you carry out the steps of the design recipe is very important. Use the following order:

1. Write a draft of the Purpose
2. Write Examples (by hand, then code)
3. Write Definition Header & Contract
4. Finalize the purpose with parameter names
5. Write Definition Body
6. Write Tests
Using the design recipe

Purpose (first draft):

;; produce the sum of the squares of two numbers

Examples:

\[ 3^2 + 4^2 = 9 + 16 = 25 \]

;; Example:
(check-expect (sum-of-squares 3 4) 25)
Using the design recipe (cont)

Header & Contract:

;; sum-of-squares: Num Num → Num

(define (sum-of-squares n1 n2) . . . )

Purpose (final draft):

;; (sum-of-squares n1 n2) produces the sum of the squares
;; of n1 and n2.
Using the design recipe (cont)

Write Function Body:

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))

Write Tests:

;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares -2 7) 53)
(check-expect (sum-of-squares 0 2.5) 6.25)
Using the design recipe (final result)

;; (sum-of-squares n1 n2) produces the sum of the ...
;; sum-of-squares: Num Num →  Num
;; Examples:
(check-expect (sum-of-squares 3 4) 25)

(define (sum-of-squares n1 n2)
  (+ (sqr n1) (sqr n2)))

;; Tests:
(check-expect (sum-of-squares 0 0) 0)
(check-expect (sum-of-squares 0 2.5) 6.25)
Tests

• Tests should be written later than the code body.

• Tests can then handle *complexities encountered while writing the body*.

• Tests don’t need to be “big”.

  In fact, they should be small and directed.

• The number of tests and examples needed is a matter of judgement.

• **Do not** figure out the expected answers to your tests by running your program! Always work them out **independently**.
The teaching languages offer three convenient testing methods:

(check-expect (sum-of-squares 3 4) 25)
(check-within (sqrt 2) 1.414 .001)
(check-error (/ 1 0) ": division by zero")

check-within should only be used for inexact values.

Tests written using these functions are saved and evaluated at the very end of your program.

This means that examples can be written as code.
Contracts

• We will be more careful than HtDP and use abbreviations.
  – **Num**: any Racket numeric value
  – **Int**: restriction to integers
  – **Nat**: restriction to natural numbers (0, 1, 2, 3, ...)
    
    In this course natural numbers include 0.
  – **Any**: any Racket value

• We will see more types soon.

*Use the most specific type available.*
**Additional contract requirements**

If there are *important constraints on the parameters* that are not fully described in the contract, add an additional `requires` section to “extend” the contract.

```scheme
;; (my-function a b c) ...

;; my-function: Num Num Num → Num

;; requires: 0 < a < b

;; c must be non-zero
```
Racket does not enforce contracts, which are just comments, and ignored by the machine.

Each value created during the running of a program has a type (integer, Boolean, etc.).

Types are associated with values, not with constants or parameters.

(define p 5)
(define q (mystery-fn 5))

In the code above, we have no idea what type is associated with the value of q without tracing through the evaluation of (mystery-fn 5).
Racket’s approach is known as **dynamic typing**, i.e. types are checked as the code is executed.

Many other mainstream languages use **static typing** in which constants, parameters and values all have specified types. Constants and parameters of one type may not hold a value of an incompatible type.

**With static typing**, the header of a function might look like this:

\[
\text{Int foo}(c:\text{Num}, g:\text{Nat})
\]

Here the contract is part of the language.

A program containing the function application \[\text{foo}(65, 100.0)\] would be illegal.
Dynamic typing is a potential *source of both flexibility and confusion*, as we will see.

Contracts are important in keeping us unconfused. However, they are only human-readable comments and are not enforced by the computer.

We can also create functions that check their arguments to catch type errors more gracefully (examples soon).

Unless stated otherwise, *you may assume that all arguments provided to a function will obey the contract* (including our automated testing).
Design recipe style guide

Note that in these slides, sections of the design recipe are often omitted or condensed because of space considerations.

*Consult the course style guide* (pages 1–19) before completing your assignments.

The marking scheme is typically

- 5% for readability / understandability (format, meaningful identifier names, etc.)
- 5% for purpose and contract
- 10% for tests and examples, combined
Goals of this module

You should understand the *reasons for each of the components of the design recipe* and the particular *way that they are expressed*. You should start to use the design recipe and appropriate coding style for all Racket programs you write.
Module 03 Summary

The Design Recipe


2. The **header** is the function name, parameters, and the define keyword. [5.1]

3. The **body** is the expression that computes the result. [5.1]

4. Three new functions for testing were introduced: **check-expect**, **check-within** and **check-error**. [12]

5. Contracts may specify the following types: **Num**, **Int**, **Nat**, **Any** with more types to come later. [13]
Module 03 Summary

Requires and Types

6. Use `requires` to specify any additional constraints on the parameters not covered in the contract. [14]

7. **Dynamic typing** is when types are checked as the code is executed. [16]

8. **Static typing** is when constants, parameters and values and the values that functions produce all have specified types. [16]
Module 4: Simple Data

Readings:

• HtDP, sections 4-5

Topics:

• Boolean-valued functions
• Symbolic data
• Strings
• Conditional expressions
• Example: computing taxes
Boolean-valued functions

A function that tests whether two numbers $x$ and $y$ are equal has two possible Boolean values: true and false.

An example application: $(= x y)$.

This is equivalent to determining whether the mathematical proposition “$x = y$” is true or false.

Standard Racket uses #t and #true where we use true, and similarly for #f, #false, and false; these will sometimes show up in basic tests and correctness tests. You should always use true and false.
Other types of comparisons

In order to determine whether the proposition “\(x < y\)” is true or false, we can evaluate \((< x y)\).

There are also functions for \(>\), \(\leq\) (written \(<=\) ) and \(\geq\) (written \(>=\)).

*Comparisons are functions which consume two numbers and produce a Boolean value.* A sample contract:

```plaintext
;; = : Num Num → Bool
```

Note that Boolean is abbreviated in contracts.
Complex relationships

You may have already learned in Math 135 how *propositions can be combined using the connectives AND, OR, NOT.*

Racket provides the corresponding *and, or, not.*

These are used to test complex relationships.

Example: the proposition “$3 \leq x < 7$” can be computationally tested by evaluating

$$\text{and} \ (\leq 3 \ x) \ (\ < \ x \ 7).$$
Some computational differences

The mathematical AND, OR connect two propositions.

The Racket and, or may have more than two arguments.

The special form and has value true exactly when all of its arguments have value true.

The special form or has value true exactly when at least one of its arguments has value true.

The function not has value true exactly when its one argument has value false.
Some computational differences

Truth tables for **and**, **or** and **not**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p</td>
<td>q</td>
<td>(and p q)</td>
<td>(or p q)</td>
<td>(not q)</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>----------</td>
<td>----------</td>
<td>----------</td>
</tr>
<tr>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>true</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td></td>
</tr>
</tbody>
</table>
Some computational differences

Some observations about and and or

• In some cases you do not need to evaluate every argument to know the answer

\[(\text{and} \ false \ x) = \text{false} \quad \text{but} \quad (\text{and} \ true \ x) = x\]

\[(\text{or} \ true \ x) = \text{true} \quad \text{but} \quad (\text{or} \ false \ x) = x\]

• This observation leads to the idea of short-circuiting, i.e. only evaluating the arguments until you know the answer.

• Because not all arguments are evaluated, and and or are special forms.
Short-circuiting

DrRacket only evaluates as many arguments of and and or as is necessary to determine the value.

Examples:

;; Eliminate easy cases first; might not need to do
;; the time-consuming factorization in prime?
(and (odd? x) (> x 2) (prime? x))

;; Avoid dividing by zero
(and (not (= x 0)) (<= (/ y x) c))
(or (= x 0) (> (/ y x) c))
Predicates

A predicate is a function that produces a Boolean result.

Racket provides a number of built-in predicates, such as even?, negative?, and zero?.

We can write our own:

```
(define (between? low high numb)
  (and (< low numb) (< numb high)))
```

```
(define (can-vote? age)
  (> = age 18))
```
Symbolic data

Racket allows one to define and use symbols with meaning to us (not to Racket).

A symbol is defined using an apostrophe or ‘quote’: e.g. ’Earth

’Earth is a value just like 6, but it is more limited computationally.

Symbols allow a programmer to avoid using constants to represent names of colours, or of planets, or of types of music.

(define Mercury 1) (define Venus 2) (define Earth 3)

Unlike numbers, symbols are self-documenting – you don’t need to define constants for them.
Comparing Symbols

Symbols can be *compared using the predicate* `symbol=?`.

```
(define home 'Earth)
(symbol=? home 'Mars) => false
```

`symbol=?` is the only function we’ll use in CS135 that is applied only to symbols.
Symbols

• Symbols are used when you want to classify values into a few categories: e.g.
  – e.g. food: ’Chinese, ’Mexican, ’Thai, ’Italian
  – e.g. courses: ’interesting, ’ok, ’boring
  – e.g. movies: ’excellent, ’good, ’average, ’bad

• You can test to see if a result is a certain symbol using
  \[(\text{symbol}=?)\]

• The contract is \textbf{symbol}=?: \textbf{Sym} \rightarrow \textbf{Bool}
Strings

Racket also supports strings, such as "blue".

What are the differences between strings and symbols?

• Strings are really *compound data*
  (i.e. a string is a *sequence* of characters).

• Symbols can’t have certain characters in them
  (such as spaces).

• More efficient to compare two symbols than two strings

• More built-in functions for strings
Strings

• A **string** is a sequence of characters, i.e. commonly called text.

• There are many move functions available for strings (compared to symbols). Strings can be
  – compared sorted alphabetically with `string<`?
  – have a length, i.e. `string-length`
  – and can be joined together, i.e. `string-append`.

• Symbols are more like multiple choice questions (a few predictable options) and strings are like essay answers (many possible options).
String Functions

Here are a few functions which operate on strings.

\((\text{string-append } "\text{alpha}\) " \text{bet}\) ) ⇒ "alphabet"
\((\text{string-length } "\text{perpetual}\) ) ⇒ 9
\((\text{string<? } "\text{alpha}\) " \text{bet}\) ) ⇒ true

The textbook does not use strings; it uses symbols.

We will be using both strings and symbols, as appropriate.
Strings vs. Symbols

Consider the use of symbols when *a small, fixed number of labels are needed* (e.g. colours) and *comparing labels for equality* is all that is needed.

Use strings when the set of values is more indeterminate, or when more computation is needed (e.g. comparison in alphabetical order).

When these types appear in contracts, they should be capitalized and abbreviated: Sym and Str.
General equality testing

Every type seen so far has an equality predicate (e.g, $=$ for numbers, symbol $=\?\?$ for symbols, string $=\?\?$ for strings).

The predicate `equal?` can be used to test the equality of two values which may or may not be of the same type.

`equal?` works for almost all types of data we have encountered so far (except inexact numbers), and most types we will encounter in the future.
When to Avoid equal?

_Do not overuse_ equal? ⇒ use a more specific predicate if the types are both known to be the same.

If you know that your code will be comparing two numbers, use \( = \) instead of equal?.

Similarly, use _symbol\(=\)? if you know you will be comparing two symbols.

This gives additional information to the reader, and helps catch errors (if, for example, something you thought was a symbol turns out not to be one).
Conditional expressions

Sometimes, the value an expression should take depends on some condition.

E.g. expressions should take one value under some conditions, and other values under other conditions.

Example: taking the absolute value of $x$.

$$|x| = \begin{cases} 
-x & \text{when } x < 0 \\
x & \text{when } x \geq 0 
\end{cases}$$
In Racket, we can compute $|x|$ with the expression

$$(\text{cond } [(\ < x \ 0) \ (\ - x)] \ \ \ \ [(\geq x \ 0) \ x])$$

• Conditional expressions use the special form \texttt{cond}.

• \textit{Each argument is a question/answer pair.}
  – The question is a Boolean expression.
  – The answer is a possible value of the conditional expression.

• Square brackets used by convention, for readability.

• Square brackets and parentheses are equivalent in the teaching languages (but they must be nested properly)

• Note: \texttt{abs} is a built-in function in Racket
The general form of a conditional expression is ...

(cond [question1 answer1]
    [question2 answer2]
    ...
    [questionk answerk]) ;; where questionk could be else

- The questions are *evaluated in top-to-bottom order*.
- As soon as one question is found that evaluates to true, *no further questions are evaluated*.
- Only one answer is ever evaluated. \(\Rightarrow\) Either the one paired with the first question that evaluates to true or the one paired with else (if it is present and reached).
Example

\[ f(x) = \begin{cases} 
0 & \text{when } x = 0 \\
x \sin(1/x) & \text{when } x \neq 0 
\end{cases} \]

\[
\text{(define (f x)}
\text{(cond [(= x 0) 0]}
\text{[else (\ast x (sin (/ 1 x)))]})
\text{)}
\]
Simplifying conditional functions

Sometimes a question can be simplified by knowing that *if it is asked, all previous questions have evaluated to false*. 

Here are the common recommendations on which course to take after CS 135, based on the mark earned.

- $0\% \leq \text{mark} < 40\%$: CS 115 is recommended
- $40\% \leq \text{mark} < 50\%$: CS 135 is recommended
- $50\% \leq \text{mark} < 60\%$: CS 116 is recommended
- $60\% \leq \text{mark}$: CS 136 is recommended
cond without simplification

We might write the tests for the four intervals this way:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) 'cs115]
        [(and (>= grade 40) (< grade 50)) 'cs135]
        [(and (>= grade 50) (< grade 60)) 'cs116]
        [(>= grade 60) 'cs136]))
```

This method *does not take into account* that if the computation gets to the second condition we know that the grade is greater than or equal to 40% ⇒ so *it can be simplified.*
cond with simplification

We can simplify three of the tests.

(define (course-after-cs135 grade)
  (cond [(< grade 40) 'cs115]
        [(< grade 50) 'cs135]
        [(< grade 60) 'cs116]
        [else 'cs136]))

These simplifications become second nature with practice.
Tests for conditional expressions

• Write at least *one test for each possible question/answer pair* in the expression.

• That test should be simple and direct, aimed at testing that answer.

• When the problem contains *boundary conditions* (like the cut-off between passing and failing), they *should be tested explicitly*.

• DrRacket highlights unused code.

• Properly tested code should have no highlights (i.e. no unused/untested code).
Tests for conditional expressions

For the example above:

```
(define (course-after-cs135 grade)
  (cond [(< grade 40) ’cs115]
        [(< grade 50) ’cs135]
        [(< grade 60) ’cs116]
        [else ’cs136]))
```

there are four intervals and three boundary points, so seven tests are required (for instance, 30, 40, 45 50, 55, 60, 70).
Tests for Boolean Expressions

Testing **and** and **or** expressions is similar.

For \((\text{and} (\text{not} (\text{zero?} x)) (\leq (\div y x) c))\), we need three test cases:

1. one test case where \(x\) is zero
   (first argument to **and** is **false**)

2. one test case where \(x\) is nonzero and \(\frac{y}{x} > c\),
   (first argument is **true** but second argument is **false**)

3. one test case where \(x\) is nonzero and \(\frac{y}{x} \leq c\).
   (both arguments are **true**)

CS 135 Fall 2019 04: Simple Data 24
Types of Tests

Some of your tests, including your examples, will have been defined before the body of the function was written. These are known as black-box tests, because they are not based on details of the code.

Other tests may depend on knowledge of the code, for example, to check specific answers in conditional expressions. These are known as white-box tests. Both types of tests are important.
Writing tests

The textbook writes tests in this fashion:

(= (sum-of-squares 3 4) 25)

which works outside the teaching languages. check-expect was added to the teaching languages after the textbook was written. You should use it for all tests.
Example: computing taxes

Purpose: Compute the Canadian tax payable on a specified income.

Examples:

Google “Canada income tax” For 2017:

- 15% on the amount in [$0 to $45,916]
- 20.5% on the amount in ($45,916 to $91,831]
- 26% on the amount in ($91,831 to $142,353]
- 29% on the amount in ($142,353 to $202,800]
- 33% on the amount over $202,800
The “piecewise linear” nature of the graph complicates the computation of tax payable.

One way to do it uses the **breakpoints** ($x$-value or salary when the rate changes) and **base amounts** ($y$-value or tax payable at breakpoints).

This is what the paper Canadian tax form does.
Examples: Calculating the tax due

• 15% on the amount in [$0 to $45,916]
• 20.5% on the amount in ($45,916 to $91,831]
• ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>$7,724.62</td>
</tr>
</tbody>
</table>

Note the notation [0 to \(x\)] means including \(x\) and the notation (\(x\) to \(y\)] means not including \(x\).
Example: Calculating the tax due

• 15% on the amount in [$0 to $45,916]
• 20.5% on the amount in ($45,916 to $91,831]
• 26% on the amount in ($91,831 to $142,353]
• ...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$100,000</td>
<td>$0.15 \times $45,916 + \   \ 0.205 \times ($91,831 - $45,916) + \   \ 0.26 \times ($100,000 - $91,831) = $18,423.915</td>
</tr>
</tbody>
</table>

Replace $45,916 and $91,831 with constants bp1 and bp2.

Replace 15%, 20.5% and 26% with constants rate1, rate2 and rate3.
Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$6,750 = rate1 \times $45,000</td>
</tr>
<tr>
<td>$50,000</td>
<td>$7,724.62 = rate1 \times bp1 + rate2 \times ($50,000 - bp1)</td>
</tr>
<tr>
<td>$100,000</td>
<td>$18,423.915 = rate1 \times bp1 + rate2 \times (bp2 - bp1) + rate3 \times ($100,000 - bp2)</td>
</tr>
</tbody>
</table>

Now let base1 = rate1 \times bp1

and let base2 = rate1 \times bp1 + rate2 \times (bp2 - bp1), etc. for base3, ...
Calculating the tax due: simplification

Using these new constants the calculations become...

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Due</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>rate1 × $45,000 = $6,750</td>
</tr>
<tr>
<td>$50,000</td>
<td>base1 + rate2 × ($50,000 - bp1) = $7,724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>base2 + rate3 × ($100,000 - bp2) = $18,423.915</td>
</tr>
</tbody>
</table>

With this plan in mind, we can begin to create the function that calculates the Canadian income tax for any income (i.e. for all five different rates).
Examples:

<table>
<thead>
<tr>
<th>Income</th>
<th>Tax Calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$45,000</td>
<td>$0.15 \times 45000 = 6750</td>
</tr>
<tr>
<td>$50,000</td>
<td>$0.15 \times 45916 + 0.205 \times (50000-45916) = 7724.62</td>
</tr>
<tr>
<td>$100,000</td>
<td>$0.15 \times 45916 + 0.205 \times (91831-45916) + 0.26 \times (100000-91831) = 18423.915</td>
</tr>
</tbody>
</table>

(check-expect (tax-payable 45000) 6750)
(check-expect (tax-payable 50000) 7724.62)
(check-expect (tax-payable 100000) 18423.915)
Definition header & contract

;; tax-payable: Num → Num
;; requires: income ≥ 0

(define (tax-payable income) . . . )

Finalize purpose

;; (tax-payable income) computes the 2017 Canadian tax payable
;; on income.
Write function body

Some constants will be useful. Put these before the purpose and other design recipe elements.

;; Rates
(define rate1 0.15)
(define rate2 0.205)
(define rate3 0.26)
(define rate4 0.29)
(define rate5 0.33)

;; Breakpoints
(define bp1 45916)
(define bp2 91831)
(define bp3 142353)
(define bp4 202800)
Instead of putting the base amounts into the program as numbers (as the tax form does), we can compute them from the breakpoints and rates.

;; basei is the base amount for interval [bpi,bp(i+1)]
;; that is, tax payable at income bpi

(define base1 (+ (* (- bp1 0) rate1)))
(define base2 (+ base1 (* (- bp2 bp1) rate2)))
(define base3 (+ base2 (* (- bp3 bp2) rate3)))
(define base4 (+ base3 (* (- bp4 bp3) rate4)))
;; tax-payable: Num → Num
;; requires: income ≥ 0
(define (tax-payable income)
  (cond [(< income 0) 0] ;; Not strictly necessary given contract
        [(< income bp1) (* income rate1)]
        [(< income bp2) (+ base1 (* (− income bp1) rate2))]
        [(< income bp3) (+ base2 (* (− income bp2) rate3))]
        [(< income bp4) (+ base3 (* (− income bp3) rate4))]
        [else (+ base4 (* (− income bp4) rate5))])))
### Helper functions

There are many similar calculations in the tax program, leading to the definition of the following helper function:

```scheme
;; (tax-calc base rate low high) calculates the total tax owed ...
;; tax-calc: Num Num Num Num → Num
;; requires base ≥ 0, rate ≥ 0, 0 ≤ low ≤ high
;; Example:
(check-expect (tax-calc 1000 0.10 10000 10100) 1010)
(define (tax-calc base rate low high) (+ base (* rate (− high low))))
```

It can be used for defining constants and the main function.
(define base1 (tax-calc 0 rate1 0 bp1))
(define base2 (tax-calc base1 rate2 bp1 bp2))
(define base3 (tax-calc base2 rate3 bp2 bp3))
(define base4 (tax-calc base3 rate4 bp3 bp4))

(define (tax-payable income)
  (cond [(< income 0) 0] ; Not strictly necessary
        [(< income bp1) (tax-calc 0 rate1 0 income)]
        [(< income bp2) (tax-calc base1 rate2 bp1 income)]
        [(< income bp3) (tax-calc base2 rate3 bp2 income)]
        [(< income bp4) (tax-calc base3 rate4 bp3 income)]
        [else (tax-calc base4 rate5 bp4 income)])))
See HtDP, section 3.1, for a good example of helper functions.

Helper functions are used for three purposes:

1. **Reduce repeated code** by generalizing similar expressions.

2. **Factor out complex calculations**.

3. **Give names to expressions**.

Style guidelines:

- Improve clarity with short definitions using well-chosen names.
- Name all functions (including helpers) meaningfully; not “helper”.
- Purpose, contract, and one example are required.
Goals of this module

You should understand Boolean data, and be able to perform and combine comparisons to test complex conditions on numbers.

You should understand the syntax and use of a conditional expression.

You should understand how to write check-expect examples and tests, and use them in your assignment submissions.

You should be aware of other types of data (symbols and strings), which will be used in future lectures.
You should look for opportunities to use *helper functions* to structure your programs, and gradually learn when and where they are appropriate.
Module 04 Summary

Boolean-valued Functions

1. The Boolean values are true and false. [2]
2. Some comparison operators are: =, <, <=, >, >=. [2-3]
3. Some Boolean connectives are: and, or, and not. [4]
4. and will produce true if all of its arguments are true. [5]
5. or will produce true if at least one of its arguments are true. [5]
6. Short-circuiting for and and or: Racket will only evaluate as many arguments as needed to determine the result. [6]
7. A predicate is a function that produces a Boolean result. [7]
Module 04 Summary

Strings and Symbols

8. Use `symbol=?` to test if one symbol equals another. [9]

9. A `strings` is a sequence of characters (i.e. text). [10]

10. There are many more operations on strings: joining, length, alphabetic order. [11]

11. Use `symbols` when you want to classify values into a few categories. [12]

12. There are many ways to check for equality: `=` for numbers, `symbol=?` for symbols, and `equal?` for any values. Use the most specific one possible. [13–14]
Module 04 Summary

Checking Conditions

13. Use cond to evaluate different expressions depending on the condition. [16]

14. Conditions are evaluated from the top to the bottom. [17]

15. When the first condition that evaluates to true is found, its corresponding expression is evaluated. [17]

16. If the condition else is reached, its expression will be evaluated. [17]

17. When a condition is reached, you know that all the conditions above it are false. You can use this fact to simplify the tests. [19]
Module 04 Summary

Testing

18. Test each \texttt{cond} question-answer pair. \[22\]

19. If there are boundary values, then the boundary value should be tested too. \[22\]

20. Tests that \textit{are not based} on the details of the code (and are typically written first) are called \textbf{black-box tests}. \[25\]

21. Tests that \textit{are based} on the details of the code, such as testing all the \texttt{cond} questions, are called \textbf{white-box tests}. \[25\]

22. \textbf{Helper functions} are used to reduce repeated code. \[38\]
Syntax & semantics of Beginning Student

Readings: HtDP, Intermezzo 1 (Section 8).

We are covering the ideas of section 8, but not the parts of it dealing with section 6/7 material (which will come later), and in a somewhat different fashion.

Topics:

- Modelling programming languages
- Racket’s symantic model
- Substitution rules (so far)
Modelling programming languages

• A program has a *precise meaning and effect*.

• A **model** of a programming language provides a way of describing the *meaning of a program*.

• Typically this is done informally, by examples.

• With Racket, we can do better.
Advantages in modelling Racket

• Few language constructs, so model description is short

• We don’t need anything more than the language itself!
  – No diagrams
  – No vague descriptions of the underlying machine
Spelling rules for Beginning Student

**Identifiers** are the names of *constants, parameters, and user-defined functions*.

- They are made up of letters, numbers, hyphens, underscores, and a few other punctuation marks.
- They must contain at least one non-number.
- They can’t contain spaces or any of: ( ) , ; { } [ ] ‘ ’ “ ”

**Symbols** start with a single quote ’ followed by something obeying the rules for identifiers.
Spelling rules for Beginning Student

There are rules for **numbers** (integers, rationals, decimals) which are fairly intuitive.

There are some **built-in constants**, such as **true** and **false**.

Of more interest to us are the rules describing program structure.

For example: a **program** is a **sequence of definitions and expressions**.
Syntax, semantics, and ambiguity

There are three problems we need to address:

1. **Syntax**: The *form or structure* we use to say things.
   
   Not: ‘*is This Sentence Syntactically Correct*’

2. **Semantics**: the *meaning* of what we say.
   
   Not: ‘*Trombones fly hungrily.*’

3. **Ambiguity**: valid sentences have exactly *one meaning*.
   
   Not: ‘*Sally was given a book by Joyce.*’

English rules on these issues are pretty lax. For Racket, we need rules that *always* avoid these problems.
Grammars

To enforce syntax and avoid ambiguity, use grammars, i.e. rules used to construct valid programs (in Racket) or sentences (in English).

For example, an English sentence can be made up of a subject, verb, and object, in that order.

We might express this as follows:

\[ \langle \text{sentence} \rangle = \langle \text{subject} \rangle \langle \text{verb} \rangle \langle \text{object} \rangle \]

The linguist Noam Chomsky formalized grammars in this fashion in the 1950’s. The idea proved useful for programming languages.
The textbook describes function definitions like this:
\[ \langle \text{def} \rangle = (\text{define } (\langle \text{var} \rangle \langle \text{var} \rangle \ldots \langle \text{var} \rangle) \langle \text{exp} \rangle) \]

There are also rules for defining constants, \texttt{cond} expressions, etc.

The Help Desk presents the same idea as
\[ \text{definition} = (\text{define } (\text{id id id} \ldots) \text{ expr}) \]

In CS 135, we will use informal descriptions instead.

CS 241, CS 230, CS 360, and CS 444 discuss the mathematical formalization of grammars and their role in the interpretation of computer programs and other structured texts.
Racket’s semantic model

The second of our three problems (syntax, semantics, ambiguity) we will solve rigorously with a **semantic model**. A semantic model of a programming language provides a method of *predicting the result of running any program*.

Our model will repeatedly simplify the program via substitution. *Every substitution step yields a valid program* (in full Racket), until all that remains is a sequence of definitions and values.

A substitution step finds the leftmost subexpression eligible for rewriting, and rewrites it by the rules we are about to describe.
Application of built-in functions

Use the rules for the arithmetic expressions to substitute the appropriate value for expressions like \((+ 3 5)\) and \((\text{expt} 2 10)\).

\[
(+ 3 5) \Rightarrow 8 \\
(\text{expt} 2 10) \Rightarrow 1024
\]

Formally, the substitution rule for a built-in function \(f\) is:

3. \((f\ v_1 \ldots v_n) \Rightarrow v\) where \(f\) is a built-in function and \(v\) is the value of \(f(v_1, \ldots, v_n)\).

Note: Rules 1 and 2 were covered in Module 02 slide 13.

Note the two uses of an ellipsis (\(\ldots\)). What does it mean?
Ellipses

For built-in functions \( f \) with one parameter, the rule is:
\[(f \ v_1) \Rightarrow v \] where \( v \) 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 \( v \) 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 \( v \) is the value of \( f(v_1, v_2, v_3) \)

We can’t just keep writing down rules forever, so we use *ellipses* to *show a pattern*:
\[(f \ v_1 \ldots \ v_n) \Rightarrow v \] where \( v \) is the value of \( f(v_1, \ldots, v_n) \).
Application of user-defined functions

As an example, consider \((\text{define } (\text{term } x \ y) (\times x (\text{sqr } y)))\).

The function application \((\text{term } 2 \ 3)\) can be evaluated by taking the body of the function definition and replacing the parameters \((x \text{ and } y)\) with the arguments \((2 \text{ and } 3)\).

The result is \((\times 2 (\text{sqr } 3))\).

The rule does not apply if an argument is not a value, as in the case of the second argument in \((\text{term } 2 (\text{+ } 1 \ 2))\).

Any argument which is not a value must first be simplified to a value using the rules for expressions.
Application of user-defined functions

The *general substitution* rule is:

4. \((f \; v_1 \ldots \; v_n) \Rightarrow \text{exp}'\) where \((\text{define} \; (f \; x_1 \ldots \; x_n) \; \text{exp})\) occurs to the left, and \(\text{exp}'\) is obtained by substituting into the expression \(\text{exp}\), with all occurrences of the formal parameter \(x_i\) replaced by the value \(v_i\) (for \(i\) from 1 to \(n\)).

Note we are using a pattern ellipsis in the rules for both built-in and user-defined functions to indicate several arguments.
Example:

\[(\text{define} \ (\text{term} \ x \ y) \ (\ast \ x \ (\text{sqr} \ y))))\]

\[(\text{term} \ (-3 \ 1) \ (+ \ 1 \ 2)) \ ; \text{apply built-in function (to 1st arg)}\]
\[\Rightarrow (\text{term} \ 2 \ (+ \ 1 \ 2)) \ ; \text{apply built-in function (to 2nd arg)}\]
\[\Rightarrow (\text{term} \ 2 \ 3) \ ; \text{replace parameters with arguments}\]
\[\Rightarrow (\ast \ 2 \ (\text{sqr} \ 3)) \ ; \text{apply built-in function (to 2nd arg)}\]
\[\Rightarrow (\ast \ 2 \ 9) \ ; \text{apply built-in function}\]
\[\Rightarrow 18\]

Note: The comments were included here to clarify each step. *On a midterm or final, do not put the comments.*
Application of user-defined constants

A *constant definition binds an identifier* (the constant) *to a value* (the value of the expression).

We add the substitution rule:

5. $id \Rightarrow val$, where (define id val) occurs to the left.
Example:

```
(define x 3) (define y (+ x 1)) y
⇒ ; substitute 3 for x
(define x 3) (define y (+ 3 1)) y
⇒ ; apply built-in function
(define x 3) (define y 4) y
⇒ ; substitute 4 for y
(define x 3) (define y 4) 4
```

Note: In order to fit this onto one slide, we have put the program onto one line and added comments, but students should avoid both of these on a midterm or final (if possible).
Module 05 Summary

Syntax, Semantics and Grammar

1. **Syntax:** The correct format (nothing unexpected or missing ). [6]
2. **Semantics:** the meaning of what we say. [6]
3. **Ambiguity:** valid sentences must have exactly one meaning. [6]
4. **Grammars** are formal rules to enforce syntax and prevent ambiguity. [7]
5. A **semantic model** of a programming language specifies a method of predicting the result of running any program. [9]
6. For Racket, our model is to repeatedly simplify the program via substitution. [9]