Before you do something, as a general rule, consider:

Why are we doing this?

and, for that matter,

What are we trying to do?
What is Computation?

A computer program is a set of instructions to complete a particular task.

Many tasks are mathematical: the computation of certain mathematical values. This will be the primary direction we move in this course.

Many mathematical questions we can answer by hand. For example:

Exercise

How many natural numbers is 12 divisible by?
Every time you see the green “Exercise” box, spend some time to carefully consider and actually do the problem.
Discuss problems in the discussion forum:
- if you are stuck;
- if you want to consider more deeply;
- for any other reason.
What is Computation?

By hand, you can determine the answer fairly easily.

But if I ask the same question of another number, I might not be able to do it by hand.

How many natural numbers is 5218303 divisible by?

By hand, this question is too difficult to solve in a reasonable amount of time. Our task in this course will be to write instructions to allow the computer to solve tasks such as these.

Instead of solving this particular problem, we will write a function that solves the problem in general, for any number.

We can test our function using small numbers like 12, 31, and 63. Once we are confident it is correct, we can use it to answer the “big” question.

(This particular question we will be able to answer in Module 5.)
**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
Attributes:

- a functional programming language
- minimal but powerful syntax
- small toolbox with ability to construct additional required tools
- interactive evaluator
- graduated set of teaching languages are a subset of Racket

Background:

- used in education and research since 1975
- a dialect of Scheme; descendant of Lisp
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.
To *design a program* we need to take the problem apart, understand it completely, and see how to instruct the computer to solve it.

- Sometimes we need only *recognize* that the problem is one we’ve solved before. Then we just use the same *technique* we used before.
- Sometimes we need to come up with a completely new solution, like nothing we’ve seen before. We need an *ad-hoc* solution.
- More usually, portions resemble what we’ve seen before, and portions are new.
With less focus on *language*, we will cover the 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, ignoring irrelevant details
Refinement revising and improving initial ideas
Syntax how to say things
Expressiveness how easy it is to say and understand
Semantics the meaning of what is said
Communication understanding others’ programs, and making your programs understandable
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)
CS 135 will progress through the Teaching Languages starting with *Beginning Student*. Follow steps 3 - 5 each time you change the language.

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*
5. Under *Fraction Style*, select *Mixed fractions*
Some simple expressions

Exercise

Guess how to translate each expression into Racket. Put them in the definitions frame. Click “Run” and make sure you understand what happens:

\[ 2 + 3 \quad 2 \times 3 \quad 44 - 2 \]

Exercise

Now translate these expressions into Racket. Run them to check your work.

\[ 3 \times 4 + 2 \quad \frac{2 + 4}{5 - 1} \quad 3\left(1 + \left(\frac{6}{2 + 5}\right)\right) \]
If I want to indicate that I am evaluating an expression, I will use a little arrow \( \Rightarrow \) between the expression and the value that it evaluates to.

For example, because \((+ 1 1)\) evaluates to 2, I could write \((+ 1 1) \Rightarrow 2\).

Similarly, \((* 3 7) \Rightarrow 21\) and \((/ 42 6) \Rightarrow 7\).

\( \Rightarrow \) is not code. It is a symbol we use to indicate what the code does.
You have plenty of experience using functions in Mathematics. Some familiar terms:

- In a **function definition** such as $g(x, y) = x - y$, $x$ and $y$ are called the **parameters** of $g$.
- In a **function application** such as $g(5, 3)$, 5 and 3 are the **arguments** for the parameters.
- We **evaluate** an expression such as $g(3, 1)$ by **substitution**. Thus $g(3, 1) = 3 - 1 = 2$.
- The function $g$ **consumes** 3 and 1, and **produces** (or **returns**) 2.
A Racket expression consists of

1. *one* open bracket: (`
2. a function name,
3. zero or more arguments to the function which may themselves be expressions,
4. *one* close bracket: `)

Even things like `+`, `-`, `*` and `/` are functions.
What is wrong with each of the following?
For each item, (1) try to predict what the problem is, then (2) run the code in DrRacket and carefully read the error message.

1 (5 * 14)
2 (* (5) 3)
3 (+ (* 2 4)
4 (* + 3 5 2)
5 (/ 25 0)
Syntax refers to how we may express things. Racket syntax is strange, but very simple.

Every expression is either a value such as a number, or of the form `(fname A B ...)`, where `fname` is the name of a function, and `A` and `B` are expressions. (More later.)

- `(* (5) 3)` contains a syntax error since `5` is not the name of a function.
- `(5 * 14)` has the same syntax error: `5` is not the name of a function.
- `(+ (* 2 4))` contains a syntax error since the brackets don’t match.
Semantics refers to the meaning of what we say. Semantic errors occur when an expression (which has correct syntax) cannot be reduced to a value by substitution rules.

- \((* + 3 5 2)\) contains a semantic error since we cannot multiply “plus” with 3, 5, and 2.
- \((/ 25 0)\) contains a semantic error since we cannot divide by zero.
Racket has many built-in functions, too many to list here. To learn about the built-in functions, we need to read the documentation.

**Exercise**

In DrRacket, select Help → Racket Documentation

This brings up the web browser, like this →

Scroll down to Teaching → How to Design Programs Languages, then Beginning Student.
Finally we see information about the functions we are interested in:

Exercise

Become more comfortable with the documentation by looking up each of the following functions:

quotient  remainder  expt  gcd
You should be quite familiar with defining functions in Mathematics.

A few 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.
- Different functions may use the same parameter name.
- Parameter order matters. $g(3, 1) = 3 - 1$ but $g(1, 3) = 1 - 3$.
- Calling a function creates a new value.
To translate \( f(x) = x^2 \) and \( g(x, y) = x - y \) into Racket, in the definitions frame, type:

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

Now we can write expressions that use these new functions.

Exercise: After defining the function above, also type in the expressions \( (f 5) \) and \( (g 46 4) \), and click “Run”. Make sure you understand what happens.

Create a few other expressions involving the newly created functions \( f \) and \( g \).
**Defining functions in Racket**

`define` is a special form. It looks like a Racket function but not all its arguments are evaluated.

It **binds** a name to an expression. This expression may use the parameters which follow the name, along with other built-in and user-defined functions.

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

**Exercise**

Use `define` to create a function `(add-twice a b)` that returns `a + 2b`.

```
(add-twice 3 5) ⇒ 13
```

Create and try out at least two other expressions that use `add-twice`. 
Functions and parameters are named by identifiers, like $f$, x-ray, wHaTeVeR.

- Identifiers can contain letters, numbers, -, _, ., ?, =, and some other characters.
- Identifiers cannot contain space, brackets of any kind, or quotation marks like `"'".
As with Mathematical functions:

- Changing names of parameters does not change what the function does.
  
  \( \text{define} \ (f \ x) \ (* \ x \ x) \) and \( \text{define} \ (f \ z) \ (* \ z \ z) \) have the same behaviour.

- Different functions may use the same parameter name; there is no problem with
  
  \( \text{define} \ (f \ x) \ (* \ x \ x) \)

  \( \text{define} \ (g \ x \ y) \ (- \ x \ y) \)

- Parameter order matters. The following two functions are not the same:
  
  \( \text{define} \ (g \ x \ y) \ (- \ x \ y) \)

  \( \text{define} \ (g \ y \ x) \ (- \ x \ y) \)
Racket also allows a special kind of value called a **constant**:

```
(define k 3)
```

This binds the identifier `k` to the value `3`.

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

The expression `(* k k)` is evaluated, giving `9`. The identifier `p` is then bound to this value.

There are a few built-in constants, including `pi` and `e`. Some programs might make their own constants, such as `interest-rate` or `step-size`.

Constants can make your code easier to understand and easier to change.
Exercise

Try out the following lines of code. If you change the order of the first two lines, what happens and why?

(define x (+ 2 3))
(define y (+ x 4.5))
x
y
“Big red trucks drive quickly” is an English sentence with correct syntax and clear semantic interpretation.

“Colorless green ideas sleep furiously”\(^1\) has the same syntax, but no clear semantic interpretation.

“Students hate annoying professors” and “I saw her duck” both have ambiguous semantic interpretation; they have multiple possible meanings.

Computer languages are designed so every program either has at most one semantic interpretation.

\(^{1}\)from *Syntactic Structures* by Noam Chomsky, 1957.
Given these definitions:

(define foo 4)
(define (bar a b) (+ a a b))

What is the value of this expression?

(* foo (bar 5 (/ 8 foo)))

Do not use Racket to evaluate the code. Try to carefully work it out by hand. Then use Racket to verify your understanding.
We wish to be able to predict the behaviour of any Racket program. We can do this by viewing running a program as applying a set of substitution rules. Any expression that does not contain an error will simplify to a single value.

For example, consider

\((* 3 (+ 1 (+ (/ 6 2) 5)))\)

Since the semantic interpretation of \((/ 6 2)\) is 3, we can simplify:

\(\Rightarrow (* 3 (+ 1 (+ 3 5)))\)

**Exercise**
Complete the interpretation of \((* 3 (+ 1 (+ (/ 6 2) 5)))\)
Now consider the following program:

\[
\begin{align*}
&\text{define} (f \ x) \ (\ast \ x \ x) \\
&\text{define} (g \ x \ y) \ (- \ x \ y) \\
&(g \ (f \ 2) \ (g \ 3 \ 1))
\end{align*}
\]

The function \((f \ 2)\) is bound to \((\ast \ 2 \ 2)\), so simplify:

\[
\Rightarrow \ (g \ (\ast \ 2 \ 2) \ (g \ 3 \ 1))
\]

Complete the interpretation of \((g \ (f \ 2) \ (g \ 3 \ 1))\)
Goal: a unique sequence of substitution steps for any expression.

Recall from before:

“Every expression is either a value such as a number, or of the form (fname A B...), where fname is the name of a function, and A and B are expressions.”

Major approach: to evaluate an expression such as (fname A B)

1. evaluate the arguments A and B, then
2. apply the function to the resulting values.

For example, to evaluate (+ (/ 6 2) 5), first we need to evaluate (/ 6 2), which gives 3. The other argument, 5, is already evaluated. The expression becomes (+ 3 5), so apply the + function to the values 3 and 5, giving 8.

Note: we do not evaluate definitions; we use definitions to evaluate expressions.
Let's all do it the same way

Evaluate arguments starting at the left.

For example, given \((* (+ 2 3) (+ 5 7))\), perform the substitution \((+ 2 3) \Rightarrow 5\) before the substitution \((+ 5 7) \Rightarrow 12\).

(Note: for functions, this choice is arbitrary, and every choice will give the same final value. But if we all do it the same way it's easier to communicate what we are doing. Some special forms, discussed later, must be evaluated left-to-right.)
Built-in function application use mathematical rules.

E.g. \((+ \ 3 \ 5) \Rightarrow 8\)

Value no substitution needed. E.g. \(7\) is just \(7\)

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

E.g. if we have: \((\text{define} \ x \ 3)\)

then to evaluate: \((\ast \ x \ (+ \ x \ 5))\)

we evaluate the arguments, starting at the left. But the first argument is the constant \(x\). So substitute (this copy only!):

\[ \Rightarrow (\ast \ 3 \ (+ \ x \ 5)) \]

Note we do not replace both copies of \(x\) in one step! We replace only the single copy.
User-defined function application  a function is defined by \( \textbf{define (f x1 x2 \ldots xn) exp} \).

Simplify a function application \( (f v1 v2 \ldots vn) \) by replacing all occurrences of the parameter \( x_i \) by the value \( v_i \) in the expression \( \text{exp} \).

For example,
\[
\textbf{define (foo a b) (+ a (- a b))}
\]
\[
\text{foo 4 3}
\]
\[
\Rightarrow (+ 4 (- 4 3))
\]

Note: each \( v_i \) must be a value. To evaluate \( \text{foo (+ 2 2) 3} \), \textit{do not} substitute \( (+ 2 2) \) for \( a \), to give \( (+ (+ 2 2) (- (+ 2 2) 3)) \).

Always evaluate the arguments first.
Beware of tricksters...

The substitution rules are very specific. If you follow them exactly, you should always get the right answer. But sometimes our intuitions will lead us astray.

Practice interpreting the code listed in the commentary.
Applying simplification rules such as these allows us to predict what a program will do. This is called tracing.

Tracing allows you to determine if your code is semantically correct – that it does what is supposed to do.

If no rules can be applied but an expression cannot be further simplified, there is an error in the program. For example, \((\text{sqr~} 2\ 3)\) cannot be simplified since \text{sqr} has only one parameter.

Racket has a feature call the Stepper that traces code automatically. We’ll discuss it as we move along.
The **scope** of an identifier is where it has effect within the program.

- Two kinds of scope (for now): global and function
- The smallest enclosing scope has priority
- Duplicate identifiers within the same scope will cause an error

```racket
(define x 3)
(define (f x y) (- x y))
(define (g a b) (+ a b x))
(+ (f 2 x) 1)
```

```racket
(define f 3)
(define (f x) (sqr x))
Racket Error: f: this name was defined...
```
DrRacket can help you identify an identifier’s scope.
Write a Racket function corresponding to

\[ g(x, y) = x \sqrt{x} + y^2 \]

((sqrt n) computes \( \sqrt{n} \) and (sqr n) computes \( n^2 \)).
Trace the program to determine what the result should be. Then remove the ; and run it in Racket to check your work:

Note: \((\text{sqrt} \ n)\) computes \(\sqrt{n}\) and \((\text{sqr} \ n)\) computes \(n^2\).

\[
\begin{align*}
\text{(define (disc a b c) (sqrt (- (sqr b) (* 4 (* a c)))))} \\
\text{(define (proot a b c) (/ (+ (- 0 b) (disc a b c)) (* 2 a))}) \\
\text{(proot 1 3 2) ⇒ ?}
\end{align*}
\]
Module summary

- You should be familiar with the discussion forum. If you have not yet contributed to discussion, start now!
- You should be able to define and use constants and simple arithmetic functions.
- Become comfortable identifying syntax errors, and expressions which are syntactically correct. Understand the syntax rules we have defined.
- Start getting used to error messages from Racket.
- Be able to trace the substitutions of a Racket program.

Before we begin the next module, please

- Read the Thrival Guide on assignment style and submission.
In this module we added the following to our toolbox:
* + - / abs ceiling define exp expt floor max min quotient remainder sqr sqrt

These are the functions and special forms currently in our toolbox:
* + - / abs ceiling define exp expt floor max min quotient remainder sqr sqrt